SF 298 MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	12/31/95	Final Report 14 Aug 95 - 13 Aug 96

4. TITLE AND SUBTITLE
Proceedings of the 1995 Monterey Workshop -- Specification-Based Software Architectures

5. FUNDING NUMBERS
ARO MIPR 197-95

6. AUTHOR(S)
Luqi

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
Naval Postgraduate School
Code CS/Lq
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITOPING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
ARO 34885.1-MA-CF

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

19960521 016

13. ABSTRACT *(Maximum 200 words)*

The purpose of the workshop is to assess current efforts, to identify results and directions for increasing the degree of automation, to build a common understanding about the integration of methods and tools, and ultimately to help bring formal methods into practical use.

The 1995 Monterey Workshop focuses on specification-based software architectures, because it is a current and practically significant large-scale problem that promises to be amenable to formalization. Some aspects of this problem are: formalizing the requirements on the components that can fit in a given slot in an architecture, developing methods for realizing or checking those requirements, formalizing types of connections, and methods for converting one kind of connection into another, and developing methods for systematically generalizing architectures. The workshop will help researchers working on formal methods for different aspects of software development to understand recent progress on formalizing other, related aspects of the problem, and to identify issues from those other areas that have direct implications for their own work.

14. SUBJECT TERMS	15. NUMBER IF PAGES
Computer-Aided Software Development, Formal Methods, Specification-Based Architectures	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

WORKSHOP ON BUILDING SYSTEMS WITH SPECIFICATION-BASED COMPONENTS

FINAL REPORT

To: U.S. Army Research Office
P.O.Box 12211
Research Triangle Park, NC 27709-2211

Date of the Report: Dec 31, 1995

Principal Investigator: Luqi, Professor
Institution: Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

STATEMENT OF THE PROBLEM

The purpose of the workshop is to assess current efforts, to identify results and directions for increasing the degree of automation, to build a common understanding about the integration of methods and tools, and ultimately to help bring formal methods into practical use.

The 1995 Monterey Workshop focuses on specification-based software architectures, because it is a current and practically significant large-scale problem that promises to be amenable to formalization. Some aspects of this problem are: formalizing the requirements on the components that can fit in a given slot in an architecture, developing methods for realizing or checking those requirements, formalizing types of connections, and methods for converting one kind of connection into another, and developing methods for systematically generalizing architectures. The workshop will help researchers working on formal methods for different aspects of software development to understand recent progress on formalizing other, related aspects of the problem, and to identify issues from those other areas that have direct implications for their own work.

WORKSHOP SUMMARY

The workshop had a series of presentations related to different aspects of specification-based architecture, and extensive discussions. Workshop papers are included in the workshop proceedings. The workshop schedule was organized as follows:

Day 1: Tuesday Sep. 12, 1995

Luqi, Naval Postgraduate School: *Welcome and Introduction*

Waugh, SEI/CMU: *Evolutionary Design of Complex Software*

DeMarco, Atlantic Systems Guild: *On Systems Architecture*

Tsai, Univ. of Illinois at Chicago: *A Knowledge Based Approach for Specification-Based Software Architectures*

Day 2: Wednesday Sep. 13, 1995

Berzins, Naval Postgraduate School: *Software Architectures in Computer-Aided Prototyping*

Goguen, Oxford University: *Algebraic Specification-Based Software Architectures*

Dampier, Army Research Laboratory: *Specification Merging for Software Architectures*

Clements, SEI/CMU: *Formal Methods in Describing Architectures*

Day 3: Thursday Sep. 14, 1995

Mok, Univ. of Texas at Austin: *Real Time Aspects of Software Architecture*

Robertson, University of Edinburgh: *Lightweight Formal Methods*

Cooke, Univ. of Texas at El Paso: *The Software Architecture for the Analysis of Geographic and Remotely Sensed Data*

Berzins, Naval Postgraduate School: *Workshop Conclusion*

The conclusions of each session are summarized by the reporters for the session, and the results are integrated into the article "Summary of the '95 Monterey Workshop" on pages 107-112 of this proceedings.

1995 Monterey Workshop

Sponsored by:
ARO / NSF / NPS

Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development:

Specification-Based Software Architectures

September 12 - 14, 1995

U.S. Naval Postgraduate School
Monterey, California

Proceedings of the 1995 Monterey Workshop

Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development:
Specification-Based Software Architectures

Held at
U.S. Naval Postgraduate School
Monterey, California
September 12 - 14, 1995

Sponsored by
Army Research Office
National Science Foundation,
Naval Postgraduate School

Workshop Chairperson
Luqi, Naval Postgraduate School

Program Committee
Valdis Berzins (Chairperson)
Jagdish Chandra
David Hislop
Helen Gill
Man-Tak Shing

Local Arrangements
Man-Tak Shing (Chairperson)
Scott Grosenheider
Ruey-Wen Hong
Osman Ibrahim
Doan Nguyen
Tuan Anh Nguyen

Attendee List

Valdis Berzins
Naval Postgraduate School
berzins@cs.nps.navy.mil

Mark Barber
Naval Postgraduate School
barbermh@cs.nps.navy.mil

Ron Byrnes
Army Research Laboratory
byrnes@airmics.gatech.edu

Jagdish Chandra
Army Research Office
chandra@aro-emk1.army.mil

Paul Clements
Software Engineering Institute
pclement@sei.cmu.edu

Daniel Cooke
University of Texas, El Paso
dcooke@cs.utep.edu

Krzsztof Czarnecki
Daimler-Benz AG
czarnecki@dbag.ulm.daimlerbenz.com

David Dampier
Army Research Laboratory
dampier@airmics.gatech.edu

Tom DeMarco
The Atlantic System Guild
tdemarco@shore.net

Joseph Goguen
Oxford University, UK
goguen@comlab.ox.ac.uk

Scott Grosenheider
Naval Postgraduate School
grosenhe@cs.nps.navy.mil

David Hislop
Army Research Office
hislop@aro-emh1.army.mil

Randall Holmes
Boise State University
holmes@math.idbsu.edu

Ruey-Wen Hong
Naval Postgraduate School
vincent@cs.nps.navy.mil

Osman Ibrahim
Naval Postgraduate School
ibrahim@cs.nps.navy.mil

Som Karamchetty
Army Research Laboratory
karamche@cs.nps.navy.mil

Luqi
Naval Postgraduate School
luqi@cs.nps.navy.mil

Aloysius Mok
University of Texas, Austin
mok@cs.utexas.edu

Doan Nguyen
Naval Postgraduate School
nguyendo@cs.nps.navy.mil

Tuan Anh Nguyen
Naval Postgraduate School
tanguyen@cs.nps.navy.mil

Bala Ramesh
Naval Postgraduate School
ramesh@nps.navy.mil

David Robertson
University of Edinburgh
dr@aisb.ed.ac.uk

Man-Tak Shing
Naval Postgraduate School
mantak@cs.nps.navy.mil

Jeffrey J.P. Tsai
University of Illinois, Chicago
tsai@eecs.uic.edu

Dennis Volpano
Naval Postgraduate School
volpano@cs.nps.navy.mil

Douglas Waugh
Software Engineering Institute
dww@sei.cmu.edu

Duard Stephen Woffinden
US Army AI Center
stevew@pentagon-ai.army.mil

Du Zhang
California State Univ., Sacramento
zhangd@gaia.ecs.csus.edu

Table of Contents

Monterey Workshop '95: Specification-based Software Architectures – Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development

Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

1. Introduction

Current software development capabilities need improvement to effectively produce software that meets users' needs. Formal software models that can be mechanically processed can provide a sound basis for building and integrating tools that produce software faster, cheaper, and more reliably. Formal methods can also increase automation and decrease inconsistency in software development.

The goal of the Monterey Workshop series is to help increase the practical impact of formal methods for software development so that these potential benefits can be realized in actual practice. Each year we focus in depth at one aspect of software development. In 1992, the focus was real-time and concurrent systems; in 1993, software slicing and merging; and in 1994, software evolution. This year's focus is specification-based software architectures.

This workshop helps clarify what good formal methods are and what are their limits. According to Webster's Dictionary, *formal* means definite, orderly, and methodical; it does not necessarily entail logic or proofs or correctness. Everything that computers do is formal in the sense that syntactic structures are manipulated according to definite rules. Formal methods are syntactic in essence and semantic in purpose. Given the motivations of the workshop, we believe this is the most appropriate sense for the word "formal" in the phrase "formal methods." We expect the ultimate main benefits of formal methods to be decision support for and partial automation of the software development process.

2. Scope

Specification-based software architectures address families of software systems with a common problem domain and common solution structures. These domains can be general, and can have many different specializations that are amenable to the same solution structure. The problem domains can have a wide variety, including scientific computing, business information systems, computer-aided design environments, real-time control systems, distributed information systems, and military applications.

We take a specification to be a formal description of the behavior visible at the external interface of a component. A specification typically describes what must happen (liveness constraints), what may not happen (safety constraints), and timing constraints.

A software architecture defines the common structure of a family of systems by specifying the components that comprise systems in the family, the relations and interactions between the components, and the rationale for the design decisions embodied in the structure. The components are subsystems that are used as building blocks in the architecture. They can have a wide variety of scales, and can themselves be defined using lower level software

architectures, resulting in hierarchical descriptions of system structure. The components are encapsulated black boxes. The component slots in an architecture are abstract in the sense that they can be filled by a variety of concrete components that satisfy the requirements of the slots. These requirements are given via the component specifications associated with the architecture.

3. The Significance of Software Architecture

Software architectures are relevant to many aspects of computer-aided software development, including automatic program generation, reuse, evolution, and systems integration.

Connecting components. Any meaningful interaction between subsystems requires a shared conceptual model that can capture the meaning of the interaction. This model serves to enable design using black-box components, because it provides designers a characterization of the behavior of the component that is independent of its realization. The most effective models are abstract in this sense. The models can also only partially constrain the behavior of a component, with the result that a slot in the architecture can be filled with a variety of components that agree on certain aspects of their behavior and differ in others. This enables a single design to provide a controlled spectrum of possible system behaviors.

The conceptual models of component behavior also enable bridging between different concrete realizations of interfaces with the same abstract meaning. Support for software architectures should include automatic means for generating bridging transformations that enable connections between components with common conceptual models but different physical realizations of the same abstract interaction, including differences in data representations and control conventions.

Relation to languages. A software architecture is based on common models of computations. The most useful of these are abstract ones, which can serve to unify a variety of concrete realizations of the same conceptual model.

Practical applications require descriptions of particular architectures. For effective automation support, these descriptions should be expressed in a formal language that embodies the computational models underlying the architecture. Different choices of models lead to different kinds of languages. Overly detailed models of computation can quickly lead to very complicated languages and architecture descriptions. A suitably abstract model is essential for simplicity and practical usefulness.

Relation to program generation. Generating programs from specifications is very difficult, and is probably not tractable in an unconstrained context. A given problem domain and a given set of general solution structures specified by a software architecture can make the problem tractable in practice. In such a situation, the program generator is not required to create new solutions to problems, but only to tailor the general solutions given by the architecture to particular instances of the problem domain addressed by the architecture. Thus the architecture defines the range of problems that can be handled by a given solution generator.

Relation to composition and reuse. One of the difficulties in creating large systems by connecting (composing) smaller systems is compatibility between the conceptual models underlying the subsystems. A software architecture defines a common conceptual model. Components designed or generated to fit the same architecture will have compatible con-

ceptual models, and thus consistent connections can be created, possibly via some bridging code to transform between concrete representation conventions. This approach can prevent severe integration problems that could in the worst case be solvable only by redesigning one or more of the components.

Reuse is subject to a similar difficulty, because independently developed components are unlikely to have completely consistent conceptual models. However, components designed to fit a given architecture can be reused without modification in the scope of that architecture. This is significant because economically effective reuse depends on reuse of components without modification.

Relation to decomposition and analysis. Complex systems are understood by people via hierarchical decomposition. An architecture contains the information about the interface conventions and the requirements associated with the component slots in the architecture that are needed to make this work on a large scale.

Relation to evolution and merging. The constraints and conventions associated with a composite design are essential to determine what can be changed without damaging a design. This is precisely the information recorded in the specification part of a specification-based software architecture. Much of the difficulties with evolution of legacy software stem from the loss of this information.

Software change merging is the process of automatically combining several changes to the same version of a software system. An essential requirement for this process is to detect all potential conflicts between changes, and to guarantee semantic integrity of the results of no conflicts are reported. The behavioral requirement information contained in a software architecture can enhance this process and enable more discriminating results. Recent results show that change merging cannot be done via a divide and conquer approach, which implies that the computational cost of change merging increases faster than linearly with the size of the system. Change merging at the architectural level has been shown to be feasible, and this approach is most promising for large systems because it appears to be computationally tractable.

Relation to networks Networks are the physical means for realizing connections between remote nodes in distributed architectures. Knowledge of the constraints and conventions associated with a connection in a software architecture can in principle be exploited by the network protocols to provide better service.

Relation to hybrid systems Some of the components slots in a software architecture can in principle be filled by components realized in hardware rather than by software. The information in a software architecture can be used to support hardware/software codesign, and can eventually lead to automatic realizations in hardware for some classes of components.

Relation to heterogeneous architectures Different components in a software architecture can in principle be implemented using different programming languages, operating systems, or hardware platforms. A carefully structured description of the software architecture that provides annotations to refine abstract architectures with physical realization attributes can effectively support generation of connections between nodes with different (and hence incompatible) physical realizations, by automatically constructing the required transformations and inserting them in the connections.

4. Workshop Summary

The workshop consists of formal presentations on the subject, interleaved with discussions to bring out implicit assumptions, clarify relationships between different points of view, and make assessments. The conclusions of each session are summarized by the reporters for the session, and the results are integrated into the article "Summary of the '95 Monterey Workshop" on pages 107-112 of this proceedings.

Towards Megaprogramming:
A Paradigm for Component-Based Programming

This paper is based on a paper published in the
Communications of the ACM, Vol.35 No.11, November 1992, pages 89-99.

Gio Wiederhold [1], Peter Wegner [2], and Stefano Ceri [3]

January 14, 1996

Abstract

Megaprogramming is a technology for programming with large modules called *megamodules* that capture the functionality of services provided by large organizations like banks, airline reservation systems, and city transportation systems. Megamodules are internally homogeneous, independently maintained software systems managed by a community with its own terminology, goals, knowledge, and programming traditions. Each megamodule describes its externally accessible data structures and operations and has an internally consistent behavior. The concepts, terminology, and interpretation paradigm of a megamodule is called its *ontology*.

The objective of this paper is to provide a conceptual framework for the study and development of megaprogramming and to identify technical problems to be addressed; we do so by proposing a *megaprogramming language*, that serves as a module interconnection language and provides the glue for joining together computations spanning several megamodules. The language allows invoking megamodules, setting them up prior to invocation by supplying repetetive arguments to them, and extracting from them data and parameters, controlling their execution, transferring and transducing data between megamodules, and achieving asynchrony and parallelism of computations.

The underlying paradigm is that megamodules provide services. Megaprogramming has the potential of substantially increasing modeling power and software productivity by conceptualizing very large software systems in terms of interactions among very large components.

1. Introduction

The problem of scaling up software engineering to handle very large software systems is recognized as a bottleneck in software engineering [CSTB:90]. The term *megaprogramming* was introduced by DARPA (see: [BoSc:92]) to motivate research on this problem. This paper proposes a framework for megaprogramming in terms of software components called *megamodules* that capture the functionality of services provided by large organizational units like banks, airline reservation systems, and city transportation systems (see Figure 1).

Megamodules realize greater abstraction power than traditional modules by stronger encapsulation mechanisms: they can encapsulate not only procedures and data, but also types, concurrency, knowledge, and ontology. Programming within a megamodule can be handled by traditional technology, while computations spanning several megamodules are specified by *megaprograms* in a *megaprogramming language*. Megaprograms provide the glue for megamodule composition and typically involve communication over networks.

[1] Computer Science Department, Stanford University, Stanford, Ca 94305
[2] Computer Science Department, Brown University Providence, RI 02912
[3] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy

```
**************************************************
* MEGAMODULE:                                    *
*   encapsulates data, behavior and knowledge    *
*   supports multiple concurrent activities      *
*   is autonomously operated and maintained      *
*   is a potential component of many megaprograms *
**************************************************
```

Figure 1: Properties of a Megamodule

In this paper we provide a framework for megaprogramming, recognizing that much work needs to be done to turn our vision into reality. The remainder of Section 1 introduces basic concepts and a running example. Section 2 outlines the general features and functionality of a megaprogramming language, which supports operations for invoking megamodules, for setting-up common arguments and extracting information, and for inspecting their state and monitoring their progress. Section 3 deals with the composition and interaction of megamodules, indicating how their compilation and optimization can take place. Section 4 describes an architecture for megaprogramming systems with libraries and databases, and deals with megamodule maintenance and life-cycle management. Section 5 positions megaprogramming relative to other work.

1.1 Encapsulation Power of Software Components

Software components provide their clients with services specified by their interface and encapsulate (hide) local structures that implement their services [Parnas:72], along the following progression:

- Functions and procedures: encapsulate statements and expressions.
- Objects and classes: encapsulate data and procedures.
- Megamodules: encapsulate behavior, knowledge, concurrency, ontology.

Functions and procedures encapsulate statements and expressions. Objects and classes realize greater expressive power by encapsulating data (instance variables) as well as procedures (methods) [Wegn:90]. Megamodules support effective hiding of domain-specific information, by encapsulating behavior, knowledge, and concurrency. A better encapsulation concept allows megamodules to model very large organizations as well as software communities having a local language, culture, and traditions.

Megaprogramming is a form of *programming in the large* that encompasses not only largeness of size but also persistence (largeness in time), diversity (large variability), and infra-structure (large capital investment):

- Size: many lines of code.
- Persistence: data survive the execution of programs.
- Diversity: of concepts, people, knowledge, traditions, software communities.
- Infrastructure: education, interfaces, tools, networks.

Traditional programming in the large, as defined in [DeKr:76], is concerned primarily with largeness of size, and does not substantially address the requirements of persistence, diversity, and infrastructure. To achieve the needed scale-up in software design and management, much current attention is placed on modeling the process and on enhancement of computer-aided software engineering (CASE) tools [Hump:88] [Blum:90].

Persistence requires great attention to the management of change. Diversity of concepts and people requires the coordination of diverse languages, multiple programming paradigms, alternate information representation, and heterogeneous components. An adequate infrastructure requires powerful tools and programming environments, as well as support of networks, system evolution, and education [BoSc:92].

The megaprogramming paradigm, modeling real-world services, is naturally parallel and distributed; it should support dynamic changes of the interface behavior and structure of constituent megamodules; for example, a bank may provide new services to its customers, while a corporation may create new manufacturing and service departments.

Megaprogramming is concerned with computations that span several software components. Its roots may perhaps be found in the work of [DeKr:76], which introduced *module interconnection languages* (MILs) in order to facilitate the construction and management of programs consisting of collections of software

components, thereby recognizing that structuring collections of modules is an essentially different intellectual activity from constructing individual modules. Recent work ([BeLP:92]) on *module interface formalisms* (MIF) provides the basis for the specification and implementation of MILs.

Megamodules require stronger module interconnection mechanisms than traditional modules because of their stronger encapsulation facilities and their support of heterogeneous interfaces and dynamic evolution. Therefore, megaprogramming languages (MPL) extend MILs' expressive power, by handling information transfer between heterogeneous modules, dynamic queries and updates by users, distributed network communication protocols, and dynamically changing the specifications of interfaces. The traditional `CALL` statement becomes overloaded when faced with this array of tasks. Work on parallel languages has already replaced a single statement with multiple segments [Rinard:92]. We generalize on that concept. We present a particular MPL that is relatively low-level as a programming language but high-level when viewed as a networking language.

1.2 Ontologies

The term "ontology" derives from logic and artificial intelligence, where it denotes the primitive concepts, terminology, and interpretation paradigm of a domain of discourse [Gruber:91]. Whereas ontologies provide a conceptual framework for talking about an application domain and an implementation framework for problem solving. They provide a partitioning of context, so that systems that would otherwise be too complex to understand, become manageable [WRB+:90, Lenat:90].

Our paradigm for megaprogramming is based on partitioning of tasks into large components, called megamodules, that provide greater spatial, temporal, and conceptual independence than traditional modules. Each megamodule should have an internally consistent ontology, to reflect a uniform conceptual framework and problem-solving paradigm. Whereas ontologies in logic are specified by systems of axioms, the ontology of a megamodule should be determined by declarations specifying the properties of named entities accessible within the megamodule. Each declaration determines an *ontological commitment* to use the defined term in the manner specified by the declaration. The collection of all ontological commitments determines the ontology. Declarations may be classified by the nature of their ontological commitment into declarations for variables, operations, behavior, and knowledge:

```
1 declare(variable, x, 3)
2 declare(operation, successor, fun(x).(x+1))
3 declare(behavior, vehicleclass, vehiclebehavior)
4 declare(knowledge, time, ontology-of-time)
```

Variables, operations, and behavior can be introduced by traditional declarations. Ontological commitments for knowledge are beyond the current state of the art and beyond the scope of this paper, but are a central goal of artificial intelligence ([Gruber:91], [Sowa:91]).

Fortunately we can separate the question of ontological commitment within megamodules from that of communication among megamodules. Since we are largely concerned with megamodule composition rather than with their internal structure, the mechanisms of ontological commitment, though important to the practical realization of megaprogramming, are not a primary concern of this paper: the specification of ontologies local to a component remains a local issue.

1.3 An Example: Transportation of Goods Between Two Cities

As an illustration, consider the transportation of goods between two cities. The general problem is that of shipping goods between any pair of cities by various forms of transportation, such as air, rail, or truck. Local transportation within each city is managed by megamodules, and intercity transportation by air, rail, and truck is managed by other megamodules. The complexity of each megamodule is substantial and the total number of megamodules can be moderately large. We examine a specific application in greater detail.

Consider a logistics application for the shipment of goods between two U.S. Navy installations, NOSC in San Diego to NRL in Washington. This Ship-Goods task can be realized by a megaprogram using three megamodules for:

1 Surface transport in San Diego from NOSC.
2 Commercial air freight.
3 Surface transport in Washington DC from one of its airports to NRL.

The megamodules (1) and (3) will be similar in structure and processing, but will use different databases. Local experts in these cities will maintain their own databases. Megamodule (2) will differ in ontology as well, as it represents and processes data according to its particular needs and according to the unique characteristics of the environment.

Megamodule (1), which deals with San Diego surface transport, will use tables that enumerate the trucks available to NOSC, the times when they are available, their capacities, and their loading facilities. It will also contain a San Diego roadmap and transit times for road-segments at different times of day. Programs within this megamodule will deal with reserving trucks, moving them to NOSC, their loading, and travel to the airport. They will also be able to produce cost and time estimates in these tasks. However, when the departure time from NOSC is not known, the estimates may have excessive uncertainty or require very large tables of alternatives for various conditions.

Megamodule (2), for planning delivery of goods via commercial air freight, is itself a substantial application. It has to be able to choose among multiple airlines, obtain schedules and tariffs, and check for available space. This megamodule may itself invoke computational codes shared by other megamodules through conventional interfaces to program libraries.

Megamodule (3), for the Washington surface-transport, will need to know routes from each of the three Washington airports. Rush hours differ among cities, and in Washington some critical roads are closed in one direction during rush hours. Geography makes a difference as well. Heuristics that are effective in San Diego, such as 'take a local road if the main road is blocked', fail when dealing with Washington bridges.

The megaprogram will then coordinate the activities of megamodules (1), (2), and (3) and present to its end-user the *best* plan for the shipment. The execution sequence of megamodules may be affected by a variety of application requirements and constraints. If the *time-of-arrival* at NRL is the critical factor, then execution should probably commence with the Washington megamodule and proceed backwards. If instead the *time-ready* of the shipment at NOSC is supplied to the San Diego megamodule, then forward computation determines the arrival time. Without such requirements, the megaprogram might focus first on the most economical flight.

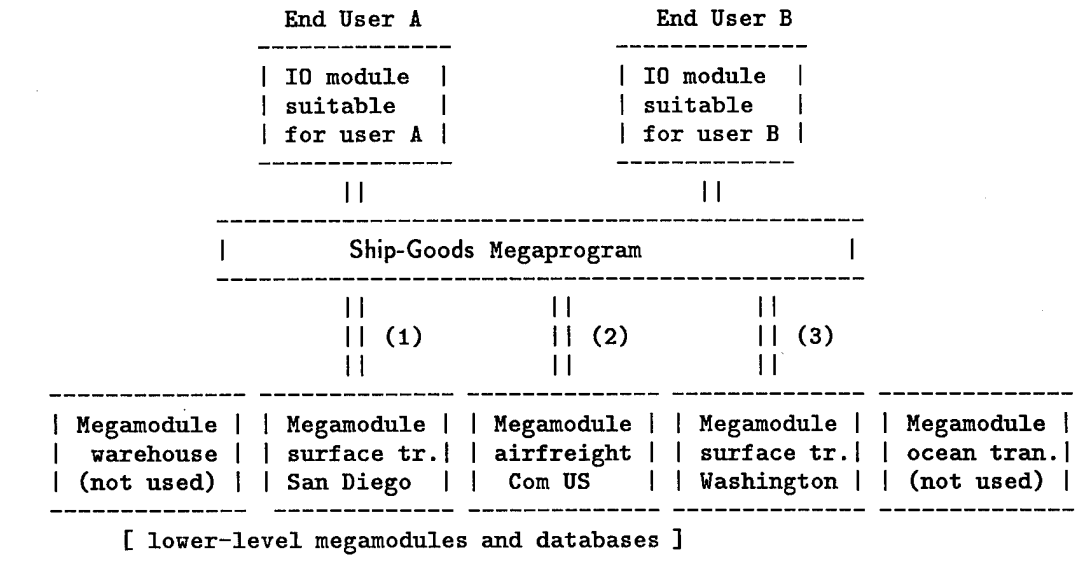


Figure 2. Example of use of megamodules by a megaprogram.

2. Megaprograms and Megaprogramming Languages

Megaprograms compose and schedule the computations performed by megamodules. Mega-programming languages must allow flexible composition of megamodules and support both synchronous and asynchronous coordination schemes, decentralized data transfer, parallelism, and conditional executions.

In this section, we propose a particular MPL [WWC:90]; many concepts presented here, though specifically related to one proposed language, are quite general. The language is used for introducing functionalities to be implemented by megaprograms and megamodules. The reader should be aware that the language and its functionalities correspond to our vision; research and development is required to turn this vision into reality. The design principles of MPL can be summarized as follows:

- Traditional CALL statements for invoking remote procedures and passing parameters, are overloaded when used to compose large systems. MPL segments their functions by separating parameter management in input and output from the invocation.
- Flexibility and asynchrony is enhanced, by allowing autonomous execution of megamodules, while the invoking megaprograms retain great flexibility, as MPL statements can monitor, inspect, and constrain the execution of megamodules and can accept partial results.
- We are inspired by our database background; databases provide services autonomously; input and output parameters of megamodules are presented through database-like schemas, and the ideas behind megamodule optimization have several common features with query optimization.

The distinguishing feature of the proposed MPL is the substitution of the traditional CALL statement, which exerts control and provides communication between software components, by three distinct statements: SET-UP, INVOKE, and EXTRACT.

a. SET-UP provides global arguments from a megaprogram to the megamodule.
b. INVOKE causes the megamodule to actually process these arguments and prepare results.
c. EXTRACT permits the megaprogram to actually extract results from the megamodule. Further statements are used for inspecting and controlling megamodule execution. The most important are:
d. EXAMINE to allow the megaprogram to check on the progress being made in obtaining results.
e. ESTIMATE, to cause the megamodule to provide a performance estimate, so that the megaprogram can judge whether invocation is likely to be effective.

To illustrate our MPL, we give a simple megaprogram that fills an order by invoking two megamodules "Producer" and "Consumer", supplying and extracting data for each megamodule, and transducing the output of the producer into the format required by the consumer. Keywords of MPL are entirely in capital letters, megamodule and megaprogram names have an initial capital letter, and datanames are entirely in small letters.

```
MEGAPROGRAM Fill-order(Producer, Consumer)
  Input product, list-of-parts
  SET-UP Producer, list-of-parts
  INVOKE Producer.product
  EXAMINE (Status) UNTIL Status = DONE
  EXTRACT Producer, list-of-produced-items
  PERFORM Transduce(list-of-produced-items, list-of-consumer-items)
  SET-UP Consumer, list-of-consumer-items
  INVOKE Consumer.purchase
  EXAMINE (Status) UNTIL Status = DONE
  EXTRACT Consumer, list-of-purchased-items
  Output cost, list-of-purchased-items
```

Figure 3. A Simple Megaprogram

Our MPL is not complex from a programming language point of view, but is higher-level than a module interconnection language. In the following, we examine the features of our MPL in greater detail; we present first the data structures for the set-up and extract operation, next the operations for megamodule interaction; then the operations for inspecting at compile-time the interfaces and content of megaprograms; and finally, those for examining the run-time status of computations. Compile-time and execution-time can easily overlap.

2.1 Data Structure Interfaces for Set-Up and Extract

The *data structures* supported by a megaprogramming declaration enable providing input and output parameters, of arbitrary complexity, between a megaprogram and megamodules; data structures are defined in the context of megamodules and are accessible by megaprograms. The MPL type system should be powerful enough to support complex data structures; therefore, it should include generalized type constructors, including records, sets, multisets, lists, sequences, updatable arrays, sparse arrays, etc.

Data structures of megamodules are designed independently; they have arbitrary internal type systems and are defined asynchronously from megaprogram definitions. Megamodules also provide self-describing `EXPORT` data structures that should be compatible with the MPL type system. However, a megamodule need not to be able to handle all of the MPL type system, its capability must only be adequate to support all of its `EXPORT` data structures.

The technology we propose to borrow and adapt here derives from database schemas, the description of database contents and formats that permits many transaction programs to share a database and obtain data selectively and associatively. Accordingly, datastructures of megamodules are declared in terms of a schema definition, but significant extension of database schema concepts will be required, since:

1 The data structures permissible in general megaprograms are more complex than the simple relations seen in most DBMSs [CCZLL:90].

2 Enough information must be provided to permit transduction of information between data structures that differ. Transduction occurs when data extracted from a megamodule are submitted to a successor megamodule: optimizations, described in Section 4, combine consecutive `EXTRACT` and `SET-UP` operations to generate a direct flow of information among megamodules.

2.2 Operations for Megamodule Interaction

These operations split the traditional "`CALL module;`" statement into three parts, to allow adequate management of large megamodules. This split also fosters asynchronous operation and parallel execution of megamodules, as depicted in Figure 4.

SET-UP The `SET-UP` statement provides information to the megamodule as a subset of the megamodule's data structure. When a set-up statement is compiled by the megamodule, the type of the MPL data structure is compared with the type of the datastructure definition previously exported by the megamodule; the megamodule is responsible for managing type coercion transformations needed for ensuring compatibility of internal data structures with the MPL type system. Problems of this translation inside a megamodule are not discussed in this paper.

The set-up of data values by megaprograms conforming to this general schema may differ. In particular:

- Many argument values may not be supplied explicitly by the megaprogram; for these default values are assumed by the megamodules. These defaults are set when `SETY-UP` has not supplied values.
- Some elements of the `SET-UP` data structures may be defined to be `PARAMETERs`. Such parameter values are specified with the actual invocation of the megamodule. During execution of the `SET-UP` statement, the megamodule retains control. This assures that the data transmission to the megamodule is synchronized properly.

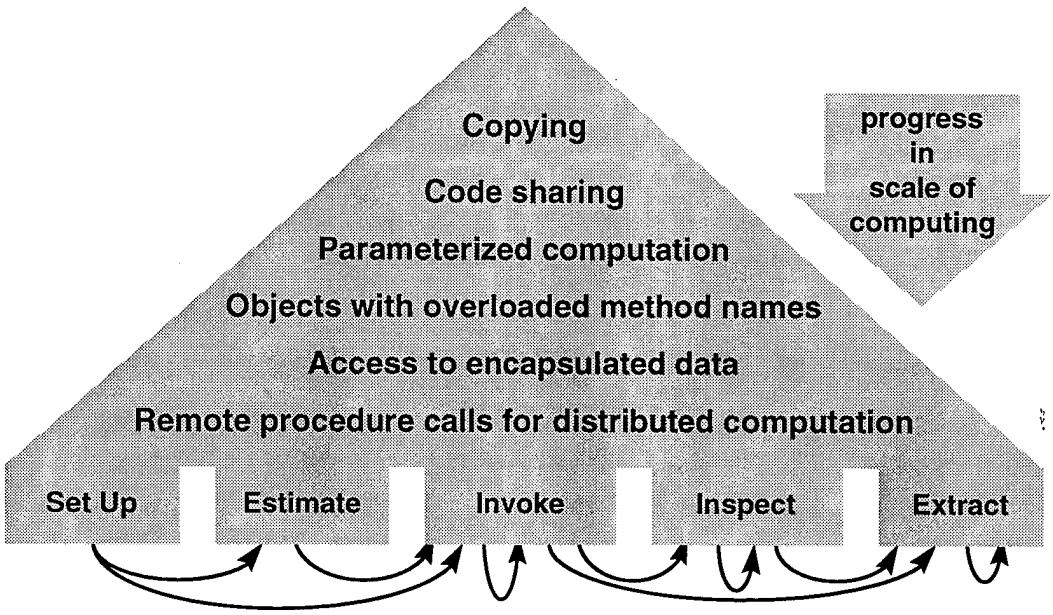


Figure 4. Modules in a mediated architecture and the scope of an application.

INVOKE The statement "INVOKE megamodule WITH parameters" causes the initiation of asynchronous execution of the megamodule with any parameter values listed in the WITH portion. The parameter values are bound to those datastructure items that were defined as parameters in the SET-UP statement. After the INVOKE statement is processed, control is returned to the megaprogram; thus, the megaprogram can schedule other work in parallel.

EXTRACT An EXTRACT statement provides megamodule results to the megaprogram. All variables that are expected to be extracted as result of the megamodule execution are made available to the MPL environment as datastructures accessed through this MPL statement. The result is fully valid when the megamodule state is DONE (corresponding to completion of the megamodule activity, as shown in Figure 3), but we place no constraints on earlier extraction of incomplete information.

During execution of the EXTRACT statement, the megamodule again retains control. This assures that in this phase, as during the execution of SET-UP, the data transmission is synchronized by the megamodule.

Note that INVOKE is typically executed asynchronously, with the parameters of the invoke being part of the activation message; megamodule operation can, however, remain independent. For instance, a megamodule may always be active and keep the EXTRACT data structure up-to-date. Then, the INVOKE message only provides a synchronization handle and the EXTRACT operation may be immediately executed. A megamodule might be active periodically based on external triggers. An EXTRACT operation then provides potentially not up-to-date snapshots; the EXTRACT operation may be executed at any time, since it does not affect the behavior of the megamodule.

2.3 Operations for Inspecting the Interface and Content of Megaprograms

The following operations, provided by megamodules, are *public* for use in a megaprogramming environment; they can be requested by megaprogrammers and their result helps in the specification and design of megaprograms. They can also be used by MPL compilers to obtain information and may be used at execution-time in interpretive situations. These operations permit megamodules to specify the data structures and defaults that they import and export; some megamodule-specific information can also be made available for inspection. Note that the information flows up, while traditionally subroutines depend for their environment on the invoking programs, and thus are difficult to reuse.

IMPORT SET-UP The IMPORT SET-UP statement causes the megamodule to export the set-up data

structures. Standard names are used to describe data structures, formats, extents, and sizes, according to MPL types. Elements of data structures may have default values which can be retrieved by an INSPECT command.

IMPORT EXTRACT The IMPORT EXTRACT statement, similar to the IMPORT SET-UP, causes the megamodule to export the extract data structure.

INSPECT Inspection of a megamodule provides information about its ontology and specific parameters; the result of an INSPECT operation does not depend on the current state of the module being inspected. Inspection could yield documentation, version information, or formal descriptions of methods and scope of applicability.

2.4 Operations for Examining the Status of Megamodules

A number of operations permit interaction of megaprograms and megamodules. Their function is to aid in execution to attain a high level of efficiency. They may, for instance, provide information that leads to a rescheduling of operational sequences of megamodule invocation. Since we assume a parallel computing environment throughout, such flexibility is essential to exploit parallel capabilities in a dynamic manner.

EXAMINE In addition to the data structures used to set-up and extract data, a megamodule maintains a number of state variables. The EXAMINE statement accesses the most critical variable, namely STATE. Reasonable values for STATE variables in megamodules are: DONE, IN_PROGRESS, IDLE, IN_ERROR, or WAITING. For instance, STATE=DONE indicates that the megaprogram can safely extract results from the megamodule. Other variables that may be provided by a megamodule include, for instance, measures of nearness to a solution for an iterative program, the number of choices being considered for a search-type megamodule, and so on. These variables may be accessed at any time for a megamodule that has been invoked. The operation of EXAMINE is synchronous with respect to the megaprogram.

ESTIMATE For megaprogramming optimization, estimates of execution cost of megamodules may be needed [Wie:92a,Wie:92b]; because of the encapsulation provided by megamodules, such estimation can only be provided from inside the megamodule. The result of execution of an ESTIMATE statement is a simple estimate of execution cost (time, dollars, result volume), with the degree of certainty of that estimate (0..1).

CONSTRAIN The CONSTRAIN statement permits the resetting, by a megaprogram, of internal megamodule parameters obtained by inspection.

LIMIT The LIMIT function constrains megamodule execution in terms of time or cost (as a surrogate for computational time spent). Not all megamodules will be able to accommodate such a request, but if they can, certain megaprogram tasks would be greatly simplified.

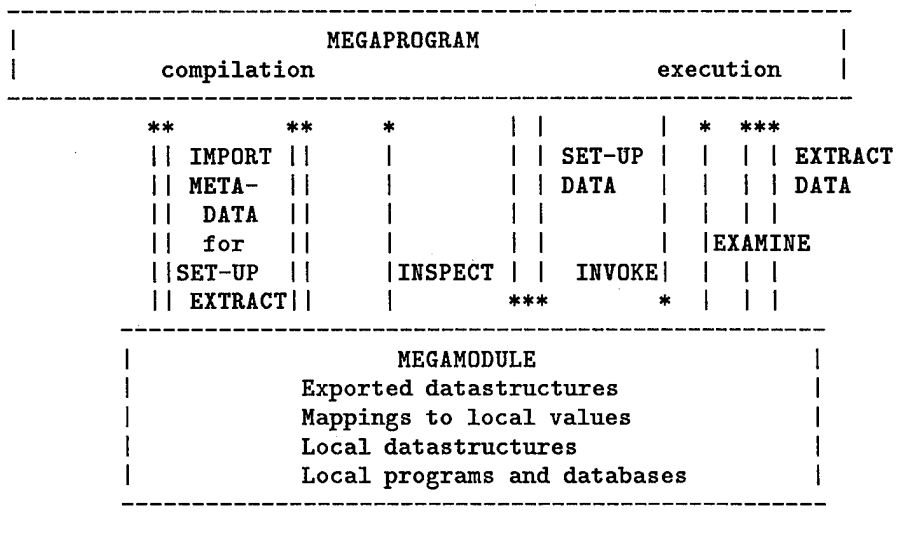


Figure 5. Information flow in megaprogamming.

3. Composition and Interaction of Megamodules

In this section we discuss the composition and interaction of megamodules as it is achieved through compilation and optimization of megaprograms, as sketched in Figure 5.

3.1 Composition and Interaction of Traditional Software Components

Function composition in traditional languages can be specified by the notation "$f(g(x))$", where f operates on the value produced by "$g(x)$". Procedure composition is realized by executing a sequence of procedure statements "$P; Q; R$" where each transforms a global state.

The composition of megamodules requires more than simple catenation. Information has to be transduced: collected, transformed, and forwarded among megamodules and input-output modules. To elucidate these MPL concepts we can compare how direct communication among programs is specified in a UNIX environment. MPL concepts can be viewed as a generalization of such UNIX concepts.

The UNIX shell plays the role of a megaprogramming language for UNIX processes. UNIX pipes provide direct communication among processes that transform a stream of input data into output data. A UNIX command of the form "$P1 \mid P2$" allows the results of the process $P1$ to be incrementally piped to $P2$, so that $P2$ can process the output stream of $P1$ in parallel with further computation by $P1$.

The generalizations provided by MPL require that we identify each MPL operation with an explicit megamodule name, since the compositions of IMPORT, SET-UP, INVOKE, EXAMINE, EXTRACT, etc., are arbitrary and any of the megamodules may be addressed. In MPL, megaprogram execution will be asynchronous, similar to what UNIX supports with the form "$P\&$". However, the megamodules remain accessible through the EXAMINE statement, so that their progress can be monitored. The combination of asynchrony and control adds much power to the megaprograming language paradigm. Research will be needed to phrase a syntactically attractive language which permits specification of the parallelism and asynchrony implicit in the megaprogramming paradigm.

3.2 Composition and Interaction of Megamodules

A simple pattern of interaction between the megaprogram and megamodules can be realized by SET-UP, INVOKE, and EXTRACT. Note that even this simple pattern of interaction improves the conventional development of applications. The SET-UP and EXTRACT data structures provide a higher level of abstraction with respect to direct linking of software modules, enabling transduction of data structures between megamodules according to mechanisms which are defined in the megaprogram. Selective specification of parameters reduces set-up and communication costs.

More complex interaction, yet still under the direct control of the megaprogrammer, is possible through the use of control structures IF ... THEN ..., whose effect is based on returned parameters from the INSPECT, IMPORT, EXAMINE, and ESTIMATE statements. By being able to assess the behavior of a megamodule in execution, it is possible to achieve important flexibility at execution time. In particular, execution of an unproductive module may be aborted and a different execution pattern may then be tried. When computational resources are plentiful, multiple similar megamodules may be started in parallel, and only the one giving the best response be retained. This approach requires that the INVOKE primitive be nonblocking, and also requires explicit time control within the MPL. The MPL will need an internal time-driven interrupt, so that it can decide when situations equivalent to TIME-OUT in conventional control of independent processes occur.

The access to a megamodule might be subject to limits, and the *megaprogram composer* might be able to check that limits are feasible through the ESTIMATE command. This merges the traditional compilation and execution phases into a complex interaction.

3.3 Compiling Megaprograms

Compilation of megaprograms is important to reduce the cost of extract, transduce, and set-up sequences among modules. This cost is especially high if the megamodules are widely distributed. For compilation and initial optimization all interactions require communication with the site where the megaprogram resides. After compilation, information can be directly conveyed among the megamodules, and the load on the network as well as on the originating site of the megaprogram can be considerably reduced.

Many of the meta-information statements, previously described as aiding the execution of an interpreted megaprogram, will be used by the megaprogram compiler to provide the input for compilization

decisions. Compilation enables *megaprogram optimization*, where the entire megaprogram is submitted to an optimization module. Such optimization would be responsible for:

- Combining consecutive **SET-UP** and **EXTRACT** operations by generating a direct flow of information between megamodules.
- Deciding the order of invocation of megamodules, in particular invoking them in the "optimal" (e.g., fastest, least expensive, ...) order.
- Introducing parallelism and asynchrony in the schedule of invocations of modules.

These optimizations might seem novel in the field of software engineering but have been successful in database systems. The information provided by **INSPECT**, **EXAMINE**, **ESTIMATE** is comparable to the information available to a database query optimizer.

The composition for our initial example, in the case that the `ready-time` at NOSC is given, might consist of:

1 Invoking the San Diego module with information about the `ready-time` and the load. Extracted from that module would be cost and time-required information, and any needed schedule and routing data. Now the earliest arrival-time of the load at Lindberg field in San Diego can be computed.

2 The air-freight megamodule is now supplied with these specifications, invoked, and similar results are extracted. The arrival-time at, say at Washington Dulles airport, for the best flight, is now computed.

3 The Washington surface module is now invoked with that time and the final arrival-time at NRL is now produced.

4 Desired output, extracted from these modules, is supplied to an output module by the megaprogram.

Many versions of such megaprograms can be envisaged. A megaprogram may iterate through the alternate airports, given as choices by the air-freight module, to optimize the Washington end of the transport task. Also, as indicated earlier, if the due-time at NRL is the critical factor, then the megamodules will be invoked in the reverse order.

```
SET-UP Surface-transport        --  set up megamodule
      ( Goods (Weight = 5000 UNIT kg,
               Volume = 30 UNIT m3,
               Maxunitsize = 5 UNIT m,
               ....)
        From (Town = San Diego, Loc = PARAMETER locn)
        To   (Town = Washington, Loc = NRL, building22)
        At   (Date = 12JUN90, Time = PARAMETER T_ready )
               ....)
INVOKE Surface-transport        --  start megamodule
        WITH (T-ready = 18:00,
              locn = (NOSC, building 17),
              ....)
```

Figure 6. Example of a megaprogram invoking one megamodule.

If minimal cost is the criterion, step 2 will likely gain priority and be scheduled first with minimal constraints.

4. Megaprogramming System Architecture

The architecture of a megaprogramming system consists of a collection of geographically distributed megamodules linked by high capacity networks; the megaprogramming environment should include a repository and dictionary, and support megamodule execution and maintenance.

4.1 Megamodule Repository and Dictionary

Information about the collection of megamodules in the megaprogramming environment is specified in a data dictionary that includes the following information about each megamodule:

1. The names of megamodules which are currently available.
2. A description of the interface of the megamodule (simply stated, some formal description of the function performed by the module).
3. Information about data to be supplied in the set-up and available for extraction, expressed as data structures in the MPL type system.
4. Internal information about megamodules; these include the programming language used by the megamodule, its internal modularization, the schemas of databases which are accessed by the megamodule, the level of consistency provided by the megamodule applications in accessing the databases, real-time constraints on megamodule execution, etc.
5. An indication of the availability and the cost of using the megamodule.

Database techniques may provide useful concepts in organizing this information and making it available to the megaprogrammer through conventional and ad-hoc query interfaces. In particular, the `IMPORT SET-UP`, `IMPORT EXTRACT`, and `INSPECT` statements in MPL may be considered as formal interfaces to this dictionary.

4.2 Support for Megaprogram Execution

The data dictionary (library) is part of a program execution environment which may be replicated at many sites, including sites associated with individual megamodules, to permit distributed specification and execution of megamodules. The responsibilities of the megaprogramming environment in supporting execution include:

1. Type checking the consistency of `IMPORT` pairs for successive `EXTRACT` and `SET-UP` operations. If the type systems are different, provide required routines for type coercion.
2. Generate executable code for megaprograms after compilation, where this code might consist of several remote procedure calls and support for data transduction.
3. Generate compiled versions of the megamodules which are used in the context of a specific megaprogram, with optimizations due to the values provided in the `IMPORT` data structure.
4. Provide late type checking for run-time parameters.
5. Handle concurrent execution and synchronization of multiple megaprograms, initiated at different sites of the megaprogramming system.
6. Deal with the notion of transaction atomicity in the context of megaprograms.

4.3 Access to Databases

We assume that databases are part of megamodules. The semantics of the database being used is completely captured within the "ontology" of the megamodule and does not need to become "public." Instead, data-structures for `SET-UP` and `EXTRACT` do become public; they will be influenced by the schemas of databases which are stored within a megamodule.

We do not focus on fully transparent distributed databases where access to any data item can be made from any site regardless of the logical fragmentation and physical distribution; this issue is not the concern of megaprogramming. If used, their locking schemes will constrain megamodule execution parallelism. Our view is consistent with autonomous and independent access; we can lower consistency requirements to match those see in federated databases [Litw:86], because we do not expect to provide comparable semantics across databases which are managed within different megamodules.

4.4 Transaction Management and Concurrency Control

We safely assume that durability (e.g., permanence of effects of committed transactions), local serializability (i.e., equivalence of each local schedule to some serial local schedule), and isolation will be provided by each local database management system. Therefore, the only issues concern global atomicity and serializability of

megaprograms, seen as large transactions operating as a unit over disperse, heterogeneous databases. This problem is quite hard, but some practical solutions have been studied; they are briefly mentioned in Section 5.3.

4.5 Maintenance

A major motivation for our approach to megaprogramming is to improve the management of maintenance tasks. The development and maintenance of each megamodule is entirely under the authority of a local software development team. This provides freedom for independent development and maintenance of each megamodule, but also confers a responsibility for providing reliable service to clients. Reliability can be ensured by globally enforced *acceptance testing* for both new modules and new versions (releases) of already existing modules. Some megamodules may have requirements for continuous execution of processes that monitor ongoing real-world activities, and cannot be taken out of service.

Maintenance is greatly facilitated by the separation between megamodules and megaprograms. Maintenance is partitioned at both the megaprogram and megamodule levels.

- Maintenance within each megamodule is done by an expert of the megamodule's "ontology." We distinguish two types of changes. *Local changes* do not affect the `SET-UP/EXTRACT` data structures, and therefore do not require megaprogram recompilation. *Global changes* affect the `SET-UP/EXTRACT` data structures, and therefore cause megaprogram recompilation.
- Maintenance of a megaprogram is done by megaprogrammers; this may be due to changes of the invocation strategies of megamodules, or might be due to global changes to megamodules.
- No maintenance operation should span two megamodules.
- While the maintenance of a specific megamodule is taking place, it is advisable to support the old version together with the new version, so that use of existing megaprograms can continue.

Observe that the rules for maintenance differ when megaprogramming:

Traditional: A maintenance programmer can touch any module of a project

Megaprogramming: A megamodule programmer can only touch a single megamodule.

5. Comparison With Related Work

Megaprogramming is primarily a software engineering activity, but borrows heavily from the technologies of databases and of artificial intelligence.

5.1 Software Engineering

The goals of megaprogramming, developing a component-based software technology for programming in the large, are also central goals of software engineering. Megaprogramming builds on earlier work on data abstraction [LisGu:86, Parnas:72], object-oriented programming [GolRob:83], and distributed computing [CoDo:88].

Scaling up from objects to megamodules goes beyond object-oriented notions of encapsulation and message passing; techniques for managing megamodules and communicating between them are qualitatively different from those for objects. Because megamodules are so much more substantial than traditional modules, techniques for object management through types, classes and inheritance are less applicable; megamodules require special database retrieval, browsing, and instrumentation methods.

Reusability is another related area in which traditional mechanisms [BiPe:89] need to be supplemented to handle the larger granularity of megamodules. We have defined megamodules as the unit of use and reuse by megaprograms in a manner that use and reuse become indistinguishable. Removing the modules from the invoking environment is critical.

Kron and DeRemer's [DeKr:76] and subsequent work [BeLP:92] on module interconnection languages and programming in the large is very influential, though megamodule interconnection mechanisms differ from traditional point-to-point communication mechanisms of object-based languages.

The need for a distinct language to manage module distribution alternatives of an application was recognized by Purtilo in his thesis and implemented in his POLYLITH software bus [Purt:90]. Our megaprogramming paradigm extends this view by proposing a compilable language for information flow control, conversion, and optimization (*transduction*), avoiding programmer involvement as system configuration and application needs change.

5.2 Databases and Transaction Processing

Work in databases has taken a number of recent directions [Silber:91] that are relevant to megaprogramming, including enhanced functionalities for data distribution and richer data models through the use of rich type systems for data definition. In this paper, we suggest that the ideal data model for megaprogramming should have a rich type system, but we deliberately don't take a position on whether it must be object-oriented; indeed, the usefulness of concepts such as object identity and generalization hierarchies with distributed data, autonomously managed by each megamodule, is currently being debated.

Databases must already deal with heterogeneity, sometimes caused by finding that distinct information resources are located at distinct sites and under distinct management. The query program has then no control over the internals of the resource representation, and very little control over the interface. These systems go by the name of *federated systems.* Languages to deal with combining information from federated systems have been developed [Daya:85] [Litw:86].

Transaction processing is another field of concern for megaprogramming. Classical transactions are minuscule tasks which only interact with each other through the database; typical workloads reach thousands of these small transactions per second. However it has become clear that not all database applications fit into this basic paradigm, so that new models of transactions are emerging, including *long transactions* [Katz:83] and *nested transactions* [Moss:85]. Long-lived transactions typically release their locks and run at a lower level of consistency with respect to full serializability. *Sagas* [Garcia:87] are particular nested transactions which achieve global atomicity but permit the autonomous commitment of component subtransactions, whose effect can be undone by compensation (typically, by firing another transaction that will reconstruct the initial state). These mechanisms may find an application in megaprogramming, where the autonomy of each megamodule is a strict requirement.

5.3 Artificial Intelligence

Independence of modules has been a long-standing concern in artificial intelligence (AI). Specifically the paradigm of independent actors and agents has captured attention from [Hewitt:73] to [Genesereth:89], proposing various models for building complex computations and knowledge bases starting from knowledge which is independently provided in different contexts. Since programming was not an objective of AI researchers, this work was focused on bringing intelligence within modules and on providing loose and flexible control of module interconnection. In contrast, we propose mechanisms for megamodule control that, though maximally flexible, are fully specified through the MPL instructions; we do not expect that megamodules propose and negotiate their own computations, as is often expected in AI models.

A broader view is taken by people working in Enterprise Integration The CARNOT project at MCC is using a single knowledge-base, within CYC ([Lenat:90]) to manage diverse resources [CWST:92]. The language they have adopted, however, was used to describe homogeneous parallel operations and does not deal directly with the diversity found.

The notion of megamodule ontology and the importance of ontological commitment (i.e., that entities within megamodules conform to their declarations) was developed in the context of knowledge representation models [Gruber:91]; megaprogramming borrows this notion in order to define the expected qualities of megamodules, which however remain out of megaprogramming control.

6. Conclusion

We have defined megaprogramming as a departure from standard top-down programming practice, which has not been able to deal well with the problems encountered when building and maintaining large software systems. Our objective has been to provide a conceptual framework for the study of megaprogramming and to identify technical problems to be addressed in developing an effective technology of megaprogramming.

Concrete application of a comprehensive megaprogramming technology along the directions outlined in this paper is clearly still far away. A number of problems need to be solved, not only related to megamodule integration, but also in the building of individual components, especially if we aim at enhancing their portability and reusability. Improving "programming in the medium" is a premise for the effective development of megaprogramming. Some of its principles can be incorporated in handcrafted systems.

Another critical factor to the success of megaprogramming is its integration with current standardization

efforts. Emerging standards for supporting remote database access and distributed commitment protocols may provide useful concepts for the technological infrastructure needed for supporting megaprogramming requirements on a computer network (based on transfer and transduction of data structures among megamodules).

Though we see an existing trend toward the design of powerful MPLs and module interface formalisms, we have to significantly raise the level of expected functionalities compared to the current state-of-art; feasibility and applicability of these ideas must be demonstrated through further research and experiments. This paper identifies a number of open research issues:

- Design of a megaprogramming language and type system.
- Design and implement the megaprogramming support environment.
- Design optimizers for megaprograms, with an increasing degree of sophistication, including the minimization of coercion and transduction costs.
- Establish trade-offs between pre-execution compilation and dynamic scheduling of megaprogams.
- Understand the role of emerging standards in the context of databases and transaction processing systems for the implementation of megaprogramming interfaces.
- Understand deeply the nature of "ontologies" as a basis for programming megamodules.
- Understand and disclose the methodological issues concerned with building individual megamodules so as to assure their reusability in the development of new applications.
- Understand and disclose the methodological issues concerned with building complex megaprogramming applications. The switch to a service from a subroutine paradigm refocuses many of these issues.

Acknowledgement

We have received useful comments on this work from Tom Dean, Tom Doeppner, Steve Reiss, and Peter Rathmann. We are grateful to the anonymous referees for their constructive comments and suggestions. A long-standing interest in new programming paradigms has been encouraged by Sheldon Finkelstein, Witold Litwin, Carl Hewitt, Yoav Shoam, and others. Work of Stefano Crespi-Reghizzi, Letizia Tanca, and Roberto Zicari contributes insight into the applicability of these concepts. The initial motivation for this formulation was provided at the 1990 DARPA-ISTO Software Enginering PI meeting, where the need for new directions in programming-in-the-large was explicated and the term *megaprogramming* appeared.

Background research for work has been partially supported by the DARPA contract N39-84-C-211. Earlier, Gio Wiederhold was supported by IBM Germany. Recent support includes a grant on coordinating updates to replicated copies on federated databases (FAUVE Contract IRI-9007753, NSF), a grant on Knowledge-based Mediators from the IBM Knowledge Systems Lab, Menlo Park, and a DARPA contract to ISI for development of standards for Knowledge-querying and -manipulation. Stefano Ceri is partially supported by the ESPRIT Projects "Stretch" and "Idea" and by the CNR Project "Logidata+."

References

[BeLP:92] F. C. Belz, D. C. Luckham, and J. M. Purtilo: "Application of ProtoTech Technology to the DSSA Program"; *Proc. DARPA Software Technology Conference 1992*, Los Angeles CA, April 28–30, Meridien Corp., Arlington VA 1992.

[BiPe:89] T. J. Biggerstaff and A. J. Perlis: *"Software Reusability"*; Addison-Wesley (ACM Press) 1989.

[Blum:90] B. I. Blum: "Modeling-in-the-Large"; draft of Chap. 3 of 'Software Engineering for Large and Small Projects', Milton S. Eisenhower Research Center, Johns Hopkins University, T.R.RMI-90-023, Sept.1990.

[BoSc:92] B. Boehm and B. Scherlis: "Megaprogramming"; *Proc. DARPA Software Technology Conference 1992*, Los Angeles CA, April 28–30, Meridien Corp., Arlington VA 1992.

[CCZLL:90] S. Ceri, S. Crespi., R. Zicari, G. Lamperti,G., and L. Lavazza: "ALGRES: an advanced database system for complex applications"; *IEEE Software*, Jul.1990.

[CoDo:88] G. F. Colouris and J. Dollimore: *"Distributed Systems: Concepts and Design"*; Addison Wesley 1988.

[CSTB:90] Scaling Up: A Research Agenda for Software Engineering, Summary of a Report by the Computer Science and Technology Board (CSTB); *Communications of the ACM*, 33:3, March 1990.

[Daya:85] U. Dayal: "Query Processing in a Multidatabase System"; in *Query Processing in Database Systems*, 1985, Springer, pp.81–108.

[DeKr:76] F. DeRemer and H. H. Kron: "Programming in the Small Versus Programming in the Large"; *IEEE Transactions on Software Engineering*, June 1976. Also in *Tutorial on Software Design Techniques*, Eds P. Freeman and A. I. Wasserman, IEEE Computer Society Press 1983.

[Garcia:87] H. Garcia-Molina and K. Salem: "Sagas"; *Proc. ACM-SIGMOD Int. Conf. On Management of Data*, S. Francisco, 1987.

[Genesereth:89] M. R. Genesereth: "Proposal for Research on Informable Agents"; Logic Group Report 89-9, Stanford University, May 1989.

[GolRob:83] A. Goldberg and D. Robson: *"Smalltalk, The Language and its Implementation"*; Addison-Wesley, 1983.

[Gruber:91] T. R. Gruber: "The Role if Common Ontology in Achieving Sharable, Reusable Knowledge Bases"; in Allen, J.A., Fikes, R., and Sandewall, E. (Eds) *Principles o f Knowledge Representation and Reasoning*, Morgan-Kaufmann, 1991.

[Hewitt:73] C. Hewitt, P. Bishop, and R. Steiger: "A Universal Modular ACTOR Formalism for Artificial Intelligence"; *IJCAI* 3, SRI, Aug.1973, pp.235-245.

[Humphrey:88] W. S. Humphrey: "Characterizing the Software Process"; *IEEE Software*, Vol.5 No.2, March 1988, pp. 73–79.

[Katz:83] R. H. Katz: "Managing the Chip Design Database"; *IEEE Computer*, Dec.1983, pp.16–36.

[Lenat:90] R.V. Douglas Lenat et al.: "CYC, Towards Programs with Common Sense"; *CACM*, Vol.33 No 8, Aug.1990.

[LisGu:86] B. Liskov and J. Guttag: *"Abstraction and Specification in Programming Languages"*, MIT Press 1986.

[Litw:86] W. Litwin and A. Abdellatif: "Multidatabase Interoperability"; *IEEE Computer*, Vol.19 No.12, Dec.1986, pp.10–18.

[Moss:85] J.Eliot B. Moss: *Nested Transactions, an Approach to Reliable Distributed Computing*; The MIT Press, 1985.

[Parnas:72] D. L. Parnas: "On the Criteria to be used in Decomposing Systems into Modules"; *Communications of the ACM, 1972.* Also in *Tutorial on Software Design Techniques*, Eds P. Freeman and A. I. Wasserman, IEEE Computer Society Press 1983.

[Purt:90] J. M. Purtillo: "The Polylith Software Bus"; Univ. of Maryland Tech. Report 2469, 1990; to appear in *ACM Transactions on Programming Languages and Systems.*

[Rinard;92] M. C. Rinard and M. S. Lam: "Semantic Foundations of Jade"; *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages* Jan. 1992.

[Silber91] A. Silberschatz, M. Stonebraker and J. Ullman (eds.): "Database Systems: Achievements and Opportunities"; *Communications of the ACM*, 34:10, Oct. 1991, 110-120.

[Sowa:91] J. F. Sowa: "Issues in Knowledge Representation" in J.F. Sowa (ed.) *Semantic Networks: Explorations in the Representation of Knowledge*, Morgan-Kaufmann, 1991.

[Wegn:90] P. Wegner: "Concepts and Paradigms of Object-Oriented Programming"; *Object-Oriented Messenger*, Vol 1, Number 1, August 1990.

[WRB+:90] G. Wiederhold, P. Rathmann, T. Barsalou, B. S. Lee, and D. Quass: "Partitioning and Composing Knowledge"; *Information Systems*, Vol.15 no.1, 1990, pp.61-72.

[Wie:92a] G. Wiederhold: "Mediators in the Architecture of Future Information Systems"; *IEEE Computer*, March 1992, pages 38–49.

[Wie:92b] G. Wiederhold: "Model-free Optimization"; *Proc. DARPA Software Technology Conference 1992*, Los Angeles CA, April 28–30, Meridien Corp., Arlington VA 1992, pp. 82–96.

[WWC:90] Gio Wiederhold, Peter Wegner, and Stefano Ceri: "Towards Megaprogramming"; Stanford University, Report No.STAN-CS-90-1341 Brown University, Report No.90-20, Oct.1990, and Politecnico di Milano, Dipartimento di Elettronica, N. 90-055.

Prospectus on Software Architecture

Douglas Waugh
Software Engineering Institute
Washington Office
dww@sei.cmu.edu

Prospectus on Software Architecture

1.0 Purpose

The purpose of this prospectus is to delineate aspects of software architecture which are pertinent to SEI's software technology transition mission, establish the scope of interest for SEI efforts, and detail the potential role of the SEI in the development and transition of software architecture technology. The role will be demonstrated through a partial listing of types of SEI-initiated activities and resulting products.

The prospectus is intended to communicate to sponsors and potential customers/users of SEI outputs which relate to software architecture. It is also intended to provide a conceptual focus for SEI technical staff whose current work relates to software architecture.Finally, it is intended to provide a common point of departure for SEI efforts on software architecture.

The first part of the prospectus discusses various aspects of software architecture. This intended to be broad and not limiting. The notion of software architecture is an evolving one, and definitive statements concerning what is and what is not included in architecture are felt to be premature and counter-productive at this time. The second part of the prospectus defines the SEI role. As contrasted with the discussion of software aspects, this part of the prospectus is intended to provide definition for SEI's work in the field and the products to be produced.

2.0 Aspects of Software Architecture

In recent years, the term, software architecture has come into vogue in the software engineering community. Although there is a core of semantic congruence among users of the term, there is also a wide difference as to just what is included in software architecture. We describe the concept we are using for our work in terms of definitions, motivations (what people are looking for from an architecture), and the activities or operations one might perform on/with an architecture.

2.1 Concepts

A software architecture contains two types of information: a representation of software system *structure* and a specification of *building codes.*

It provides *abstraction* and *codification* of structure and supports multiple structural *views* and *models*. Views provide varying representations, sequences, and slices of a single architecture. Views can be used to provide such commonly used displays as *control flow* and *data flow*. The software architecture defines software *components* and the *relationships* and *interactions* among components.

Building code information includes such things as *properties* (functional and non-functional), *standards*, *rules*, and *style*.

The ultimate aim of software development is a concrete representation of the software that executes within a system to provide desired function. There are many levels of abstraction, however, that are involved in getting to this ultimate concrete operating software. In fact, we can think of the software development process as a continuum of representations from abstract to concrete. Architecture as an artifact might be produced at any point along that continuum to enhance communication among builders, buyers, and users at that point. In order to articulate the architecture concept, it is useful to pick 3 points along the continuum and describe them using a Department of Defense system procurement as an example.

The first point we will call the Architecture for an *Application Domain*.For example, the DoD decides for a given application domain -- say Command and Control -- to produce a software architecture to represent an abstraction of C2 that is common to all of DoD's command and control. DoD might assume ownership of the architecture and use it to influence commonality across various DoD programs with respect to C2 and to communicate a structure to potential providers of C2 systems. Indeed, the USAF Electronic Systems Command has done just that with the PRISM project.

The second point on the continuum we will call the Architecture for a *Product Family Domain*. In this case, the provider of software systems to the government (a vendor or contractor) has decided for business reasons to create a product line that addresses all or a part of the C2 application domain. The abstraction of this line of products is represented in a software architecture.

The third point is the Architecture for a *System*. In this case, a contractor creates a software architecture in response to a specific DoD procurement. It is, like the other architectures, an abstraction in that many designs and implementations may result. It should comply with the appropriate application domain architecture and may make use of product family domain architectures.

These three are all considered software architectures. They will differ in degree of generality, intended use, specified constraints, etc.

2.2 Motivations for Software Architecture

One way to discuss the concept of software architecture is to consider what one wishes to accomplish through the use of an architecture.In general, people are motivated to develop and use software architectures by the following:

- to communicate effectively to the various participants in a development activity what to build, what it must do, and how to build it.
- to enhance the ability to adapt to change both during development and post deployment.
- to support analysis and prediction prior to creating concrete solutions.

- to move from expensive point solutions to solution families (analagous to product lines from the software vendor community) that emphasize common parts and reuse.
- to give guidance and controlling information analogous to commercial building codes which benefit the community such as standards to be followed, preplanned interoperability requirements, use of Commercial Off The Shelf software products, etc.

2.3 Architecture-Related Operations

This concept of software architecture includes many different operations with/on architectures. The list of operations includes but is not limited to:

create and maintain architectures

ensure completeness and consistency of an architecture

evaluate relative merit of alternative architectures

use an architecture to analyze system properties

test compliance of design or implementation with architecture

selectively view the architecture according to various criteria

predict development parameters as well as characteristics of the developed system

analyze the impact of an intended change to a system at any point in its development/ operation

track properties throughout lifecycle

validate conformance to rules, standards, style, etc.

3.0 Scope of Interest

Our focus is on ***software architecture***. The term, architecture, is also applied to hardware where it normally depicts a configuration of hardware boxes or components, indicating interfaces among them as well as hierarchical groupings into sub-assemblies, assemblies, and systems. Some references to architecture discuss an operational organizational structure for the system. It would indicate various operating elements and installations along with communication paths among them. Neither of these types of architecture is considered within the scope of this activity, although they both may be very important to the ultimate success of the system being developed.

The terms, *information architecture* and *system architecture* are also in common use. Information architecture is very similar to software architecture except that rather than focusing on software components of a system it focuses on information components. It is possible to consider information architecture as one of the views supported by the software architecture. That is, the architecture can be viewed and analyzed from an information perspective. System architecture usually suggests a representation of the allocation of function to the hardware and the software components of the system. This is not per se the focus of this effort. However, there are many instances in which an understanding of the system aspects is needed in order to do an adequate job on the software. For example, con-

sider an architecture-driven reengineering of legacy software. As reverse engineering produces an understanding of the legacy, it may become obvious that the reengineering transformation desired can be better accomplished by revisiting the original allocation between hardware and software.

4.0 SEI Roles, Activities, Products

The SEI recognizes that much good work is going on in the area of software architecture technology and the use of software architectures in practice and wishes to add value to this work as it relates to software development practitioners. To add value, the SEI assumes certain roleswith respect to software architecture:

- provide "corporate memory" for technology development and use;
- create common universe of discourse to promote understanding and interchange;
- provide practitioners with capability to make disciplined, informed choices;
- create and maintain a software architecture community;
- assist in narrowing the gap between state-of-art and state-of-practice;
- identify areas requiring additional research.

A detailed plan for SEI's initiative in this area is forthcoming, but a preliminary definition of activities and resulting products follows.

1. Facilitate assessment / evaluation of technology that supports the representation and analysis of software architectures.

   SEI will compile criteria and methods to be used in assessing and evaluating software architecture technology. Also to be produced and maintained are state-of-art and state-of -practice data.

2. Enable the analysis and evaluation of specific software architectures.

   The SEI will define a framework that will provide improved analytical methods for architectures.

3. Baseline data on software-intensive system architectures

   The SEI will capture data concerning the use of architecture and will evaluate the impact of its usage.The Institute will make the data and the evaluations available as experience history that will improve predictability.

4. Promote community awareness and provide services of common concern

   In this activity, we list products such as definition of common terminology and concepts, on-line catalogues of applicable information, and stimulation and sponsorship of common interest groups.

5. Provide technology transition assistance

The SEI will assist in narrowing the gap between state-of-art and state-of-practice and in promoting wider adoption of best practices as appropriate.

6. Provide focus for needed research

   The SEI will help to focus feedback to the research community regarding gaps in the current technology as determined by state-of-art and state-of-practice analyses. A periodically updated report on status and needs will be produced.

5.0 Summary: What Can You Expect from the SEI?

Using the architecture concept and the area of interest oultined in this prospectus, the SEI intends to:

- provide semantic models and taxonomies that help precipitate terms of reference for software architecture concepts;
- provide frameworks that will enable others to analyze and evaluate software architecture technology as well as the architectures themselves and to make intelligent design choices;
- shorten the time between the creation of applicable technology and its widespread usage by the practicing community; and
- obtain and maximize leverage from others' efforts.

ON SYSTEMS ARCHITECTURE

BY TOM DEMARCO

In this brief reflection, I shall comment on systems architecture and the effect of its absence, lay down some definitions, and then end with a discussion of the odd dynamic that impedes the successful architecting of most software. It is in this last section that I address what I believe to be our major architectural problem: since we attribute most architectural failure to the wrong cause, we are not making much progress in the area. Just to give you a hint of what is to come: It is my thesis that though we think of the inhibitors of successful architecture as entirely technological, they are much more likely to be economic, political and sociological.

ARCHITECTURE AND ITS ABSENCE

The architecture of a complex software system is analogous to the infrastructure of a highly evolved social system or biological organism. This parallel is useful to my purposes here because it highlights that the absence of an explicit architectural act may nonetheless result in something we all identify as an architecture. Early London, as an example, had been the beneficiary of no grand plan, and yet a workable infrastructure was there for all to see: goods arrived, people were induced to perform needed work, wastes were carried away, water flowed to where it was needed. The varied interests of many participants combined to create a force of evolution that provides constantly improving infrastructure. No single mind ever conceived to that result, yet it happened. This is similar to what Adam Smith identified as the basic mechanism of the market: many 'invisible hands' acting together to lead to a useful outcome.

On a recent trip to India, I encountered another example of an architecture without an architect. In the city of Bombay and surroundings, an unusual set of cultural circumstances have inhibited the creation of sufficient small restaurants to serve the lunch-time needs of the working population. The result is that, since Bombay's great explosion of wealth and prosperity, the city's workers have continued to provide their own lunches from home. Lunch is a hot meal in India, and so a network of delivery men has sprung up to transport hot meals from the worker's home, (where they are prepared by family members,) to the worker's office. These delivery men are called in the local vernacular, "box-wallahs."

The box-wallah provides an insulated box to hold the meal, a labeling system to get it to its destination, a pickup and delivery service, and the same service in reverse during the afternoon to return the containers and utensils back

to the home. The actual delivery involves several transfers of both boxes and funds; by the time a box arrives, it may have passed through the hands of half a dozen box-wallahs. The Bombay train stations around noon are crammed with literally thousands of box-wallahs carrying racks of boxes. To make matters a bit more complex, the box-wallahs are in general illiterate. That means the labeling scheme has to be understood by people who can't read. By the way, the same code serves in both directions: it gets the box to the office in the AM and then back to the home in the PM. The Bombay box-wallahs deliver more than 500,000 meals a day.

Now imagine you had been the architect of Bombay's box-wallah system. I list just a few of your considerations in laying out the scheme:

- what is the labeling scheme?
- what is the equitable payment scheme that assures all of a given box's handlers get adequately remunerated?
- what happens when one of the box-wallahs doesn't show up on a given day?
- how does change of location get handled?
- how shall routes be allocated?
- under what circumstances are routes changed?
- how does each transfer work?

As you can see, this is anything but a trivial problem.

It's amazing that the system can be made to work at all, much less as well as it does. What's more amazing still is that there is no company that runs the box-wallah network, no ownership, no management and no designer. No single mind ever thought out the answers to any of the questions I listed above, nor to any of the other details that are necessary to make the network function. The entire system simply evolved.

EVOLVED OR "ORGANIC" ARCHITECTURE COMPARED TO EXPLICIT ARCHITECTURE

Just because there is no architect and no explicit architectural act does not assure that there will be no architecture. There is always an architecture, a good one or a bad one, but there always is one. Over time, the architecture of any system tends to improve. This is a bit of a surprise, since we also know that the micro design of a system tends to worsen over time as many individual acts of maintenance make successive change more and more difficult.

The human body has evolved to have an admirable architecture. Similarly, an evolving software system can begin to exhibit some semblance of architecture. Take for example the operating system called DOS, one that is famous for its near total lack of original architecture. At the beginning, it was a strapped together set of concepts and code fragments from past operating systems, a bit of CPM, a bit of old Digital Equipment OS technology, some tape and glue. It resembled an airplane made by strapping wings onto a jet engine and tying a wooden chair on top for the pilot to sit in.

Over time, though, things changed. The very first TSR program was a complete kludge, just another element strapped on. But by the time the second and third TSR had come along, and the 200th and 300th, DOS had begun to evolve to accommodate the arrival of the 301st and subsequent TSRs. Similarly the scheme of I/O channels, largely neglected in the original PC design step (if there was one), has been rectified over time. Today, the PC can accommodate serial and parallel I/O, SCSI, and various networking interfaces. The invisible hand of the marketplace has supplied all the architecture that is in the modern descendent of the original IBM PC.

While this organic architecture does happen on its own, it is hardly what we should be trying to achieve. It takes forever and results in numerous and expensive dead-ends along the way. Worse, it may end up serving its own needs rather than ours. The opposite of an evolved architecture is an explicit original act of architecture that provides successful initial direction to a system or family of systems. The best example of this that I know is the system family that began at Xerox and ended up at Apple Computer, the family that I shall call "Dolphin-Diablo-Macintosh."

A TRIUMPH OF ARCHITECTURE

The Dolphin-Diablo-Macintosh family has been a triumph of purposeful architecture. The family has had, from its beginning, a huge capacity to respond to changing needs. Its ability to accommodate change has surprised even those who were responsible for its architecture. Take as one tiny example the multiple screen facility that was implemented on the Macintosh in the early eighties. No one had explicitly foreseen the possibility that a Mac might someday have more than one display, but when the need was identified, the implementation proceeded rapidly and without pain. The nearly immediate result was a "display space," sometimes served by a single screen and other times by an array of up to nine. Though the screens may be of different dimensions, different aspect ratios, different resolutions, and different color depths, the scheme works flawlessly. Today I can drag a desktop icon across my whole display space, from my RGB monitor, through a multisynch monitor to my television screen, and drop it there without ever worrying that it may get lost in a crack between the screens, or that any one of my screens won't know how to display the image.

It is a trivial matter to make a design that can accommodate some change that was explicitly expected by the designer. But how do we design to accommodate *unexpected* change? This is the central technological challenge of system architecture. While it is a most important architectural concern, I shall offer no specific insight on the subject beyond a pointer to one marvelous book: *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. [Reading, MA: Addison-Wesley, 1995]. In this landmark work, the authors introduce the idea of a *pattern* or reusable abstraction. For an investment of two hours in reading Gamma et al., I predict you will come away persuaded as I was that patterns are the key to

successful architecture and an answer to the question about how to design to accommodate unexpected change.

I make this claim in order to assert (at least implicitly) that the how-to's of system architecture are not beyond us. That is an essential point for this discussion, because as long as we remain convinced that some new breakthrough is required before we can begin a regular practice of architecting new systems, we will never get on to the real problem. Before proceeding to what I believe to be the real problem, I shall take just a few lines for some definitions:

WHAT IS THIS THING CALLED ARCHITECTURE?

An *architecture* is a framework for the disciplined introduction of change. This is also a pretty good definition of *design*. The difference between the two is that design, as we commonly use the word, applies to a single product, while architecture applies to a family of products.

Design is always concerned with capacity to accommodate change. This is true even of the extreme example of a system which is to be run once and then thrown away. It still makes sense to design such a system, because it still has to be able to accommodate one change, specifically the change that is the implementation process. (Here I portray implementation as a change from concept to reality.) A well designed system accommodates that change easily and a bad one doesn't. Just as a test of our definition, consider an even more extreme example: a system that is never to be run and need not even be built. I assert that there is no such thing as a "good" design for such a system. The concept of design makes no sense except in the context of change.

The absence of a thoughtful architectural act assures that there is no initial accommodation to the changes that will propel the product from version to version, or the change that allows the essence of one product to be carried over into other members of a product family. As Gamma, Helm, Johnson and Vlissides point out, our problems in achieving meaningful reuse are tied to this absent architectural act.

If architecture is so important and, if as *Design Patterns* points out, there is a workable architectural methodology, then why on earth don't we do it?

THE REAL PROBLEM

The real problem is that architecture costs money. Before you tell me that you already knew that, that of course it costs money, that "there is no such thing as a free lunch," let me give you pause by indicating *how much* architecture costs. The Dolphin-Diablo-Macintosh family probably cost Xerox PARC some $30,000,000 in its initial investment. Part of this it recouped when Apple purchased some of the rights to the desktop concept in the late seventies. Considering that transfer payment and the work of its own architecture group, Apple had invested about $25 million in architecture before beginning the Lisa project. Today, if all architectural investment were capitalized, Apple would have it on the books as more than $125 million. How does that compare to any

single development project at Apple that makes use of the architecture? Consider the Power PC implementation which occupied some 300 people for most of two years. The project cost around $60 million, supported by a capital investment of $125 million. The sobering fact is that a decent architecture costs as much as two full implementations. Your very first implementation will cost you about three times more with an architecture, compared to throwing the product together without paying for a single architectural thought. What we're talking here is real money.

In my consulting practice I see one organization after another engender an architecture group because they realize that architecture is the key to product family reuse, and thus a desirable goal. But then the architecture group is funded only as a part of the first implementation project, and that project's budget is set in the usual fashion for our industry, i.e., in the mid-range between Impossible and Highly Unlikely. In other words, the architecture group is a sham. It has a charter, but no funding. After a momentary digression on reuse, it is treated as an adjunct to the project and driven by the same dynamic as all the rest of the project, getting the product out the door as quickly and cheaply as possible. It is no surprise that no useful architecture ever comes out of such a group. Organizations that pay the cost of architecture get what they pay for and organizations that don't don't get anything at all. My experience is that the enormous majority of architectural efforts in our industry are in this latter category.

While I'm at it, there are two further inhibitors to architectural success, and these two are also strictly non-technical:

- Early Overstaffing: Projects that have too many people on board when the critical architectural work needs to be done are forced to hurry on to the kinds of tasks (coding and testing) that can use up all those people. Early overstaffing is a sign of an organization's unwillingness to invest time in architecture.
- Production Mentality: The more an organization tries to force software development into a production niche, the more wary it is of a research-like endeavor such as system architecture. Thus one well-known networking company runs a separate project for each platform, rather than develop a single cross-platform architecture. Doing the same project over and over again, building identical functionality on different platforms, is a production activity. The company can even achieve a high CMM level in doing this kind of work. Building a single cross-platform architecture, on the other hand, is a damned hard R&D project, one that scares the company silly. Such a project might even lower the organization's CMM level. So the company just can't do it. Fortunately the invisible hand of the marketplace will provide some competitor who can, and the production mentality company will go out of business (probably with CMM level 3).

A GRIM HISTORY LESSON

The best architectures of past have all been for naught. PARC failed to recoup its full investment. Xerox's "office of the future" never made a nickel for the company. Apple has not prospered. IBM's 360 hardware architecture —a marvel of the time— and its ambitious System Network Architecture (SNA) did not herald the beginning of a new prosperity, but the end of an old one. It seems that the companies that invested in meaningful software architecture have all declined. Why is this?

Before concluding that architectural projects are the cause of this decline, I urge you to look a bit further. Architecture is a big investment. Companies that make big investments are usually successful companies, acting sensibly to become more successful in the future. Xerox's decline was not the result of its architecture, but of the success that paid for that architecture. Specifically, that success sparked a ruinous assault by the Justice Department against Xerox and eventually resulted in a consent decree that forced the company to give away its principal patents, X-1 through X-9. The main beneficiaries of this patented technology were Cannon, Sharp, Minolta and a host of small Asian printer and copier companies.

Similarly, the success that enabled IBM to develop the 360 and SNA, also set off anti-trust action against it. The result was the company was forced to release its products for "cloning," something that was unheard of only a few years before. Cloning is stealing someone else's intellectual property, something that is specifically illegal in most of the world. But IBM had to permit it, first with the 360 and then with its PC line. The beneficiaries were the Asian cloners.

Apple was also a victim of the anti-trust action against IBM. In a head to head battle against IBM, Apple might have fared well, but it could not compete against the cloners who had been given IBM's intellectual property for free.

Today, one of the most auspicious architectural endeavors is that undertaken by Microsoft. It has already produced Ole, the Foundation Classes and a host of other architectural wonders. And sure enough, the Justice Department is right there again. Who would be the beneficiaries, I wonder, if Microsoft were broken up, forced to release its investment to the public sector for the free use of all? I suspect it would again be foreign competitors.

While those who have invested most prominently in software architecture have often not reaped the expected benefit, those who have failed to invest where such investment was required have fared even worse. A striking example of this is the FAA's attempt to re-implement our national air traffic control system, the NASPLAN projects. A total failure of architectural thinking has assured that these projects had almost zero chance of success. Indeed, by any way of reckoning, the entire effort has been a fiasco. There is no redesigned Air Traffic Control System on America's horizon, and most people who have been involved in the effort in any way have acquired a psychopathic fear of flying.

REPRISE

System architecture is expensive, but probably not as expensive as its absence. Today we have the capacity to build successful architectures, but often not the will. Our largest and most successful software companies have been the rare exceptions, and these have maddeningly been tageted by Justice Department actions that have resulted in loss of payback on their investments. The problems that have most often hampered architectural efforts have not been technical, but political, economic and sociological.

A KNOWLEDGE-BASED APPROACH FOR SPECIFICATION-BASED SOFTWARE ARCHITECTURES

Jeffrey J.P. Tsai
Distributed Real-Time Intelligent Systems Laboratory
Department of Electrical Engineering and Computer Science, (M/C 154)
University of Illinois at Chicago
851 South Morgan Street
Chicago, IL 60607-7053
E-mail: tsai@eecs.uic.edu

October 30, 1995

Abstract

The concept of software architecture has recently emerged as a new way to improve our ability to effectively construct large-scale software systems [Garlan95, Luqi95]. This paper presents a knowledge-based approach to construct a specification-based software architecture. In this framework, a formal requirements specification language, FRORL (Frame-and-Rule Oriented Requirements Language), using hybrid knowledge representation techniques is developed. Since errors in requirements can cost ten or more times to fix in maintenance phase, various verification mechanisms will be used to reduce errors in the system's requirements specification. The language also supports the feature of operational interpretation. Through the execution of a requirements specification, users can validate functional and non-functional requirements of the system. Debugging algorithms have been also implemented to correct any errors which might occur during the development of the specification. In addition, a new parallel execution model is established to speed up the verification and validation process. After the specification has been correctly evaluated, the specification can be transformed into target systems.

1. Introduction

The concept of software architecture has recently emerged as a new way to improve our ability to effectively construct large-scale software systems [Garlan95, Luqi95]. This paper presents a knowledge-based approach to construct a specification-based software architecture to support software development of real-time distributed systems. In this framework, a formal requirements specification language, FRORL (Frame-and-Rule Oriented Requirements Language)

[Tsai88, Tsai92a, Tsai93], using hybrid knowledge representation techniques is developed. Various verification mechanisms will be used to reduce errors in the system's requirements specification. The language also supports the feature of operational interpretation. Through the execution of a requirements specification, users can validate functional and non-functional requirements of the system. Debugging algorithms have been implemented to correct any errors which might occur during the development of the specification. A new parallel execution model is also established to speed up the verification and validation process. After the specification has been correctly evaluated, the specification can be transformed into target systems. In addition, a methodology is developed to guide the developers to focus on the provided guidelines in order to construct the formal specifications represented in FRORL from rough and vague concept down to clear and solid formal representation model [Tsai93]. During the evolving phases of software development, the software development system can provide rapid prototyping through querying facilities in order to realize the expected behavior of the specifications represented in FRORL. The developer can also have the software development system to perform properties checking, to do specification debugging, or to transform the specifications of FRORL into various target programming languages. This paper presents techniques and tools developed in the proposed knowledge-based software engineering framework.

2. Frame and Rule Oriented Requirements Language – FRORL

FRORL was designed to facilitate the specification, analysis, and transformation of real-time distributed systems [Tsai92a, Tsai93]. To provide a friendly user interface, FRORL uses the concepts of frames and production rules to represent the structure and behavior of a software system [Tsai88, Tsai92a]. A frame is a node in a circle-free hierarchical network. The links among frames realize abstract relations among them. There could be several tree forests of frame hierarchies. A node on the top level represents the most general concept of the family of all of its descendants, while nodes on the lower levels of the frame hierarchy stand for more specific instances of those concepts. Because of their object-oriented taxonomic structure, software systems that specify the use of frames are easy for people to create and comprehend in more natural way. Production rules allow the derivation of additional information from knowledge explicitly expressed in the network of frames. The frame representation is used for structural description of domain objects. The production rules specify the behaviors of the objects and interactions/relationships among object frames. To deal with incompleteness nature of users' requirements and provide a solid theoretical foundation, the semantic of FRORL is defined through the non-monotonic/temporal extension of Horn-clause logic, and this extension to Horn-clause logic will allow exceptions and inheritable relations for FRORL [Wei94]. Also, we have proven this logic to be complete and consistent [Tsai93]. The semantics of the logic presented in [Tsai93] will serve as the declarative interpretation of a FRORL specification.

FRORL also supports the feature of operational interpretation. The production rules serve to guide the execution mechanism in the application domain. The operational interpretation of the FRORL representation can realize and validate the system behaviors against the informal conceptual requirements models. The declarative interpretation of the prototype specifications is given in terms of minimal Herbrand models. The full machinery of Horn-clause logic is available to perform properties checking of the specifications should be continually tested and modified until the original requirements and constraints are satisfied by the prototyping behaviors of the declarative representations in FRORL.

Objects and activities are the modeling primitives of FRORL. Each object stands for some entity in the world domain begin modeled. The changes taking place in the world and relations among the entities of the world being modeled are represented as activities in requirements model. Each object or activity frame has certain properties, assertions, or constraints associated with

it. The information related to objects and activities is represented by the frame representation. The structure of the executable specification written in FRORL is modularized into different activities. Because of its modularized property, reusability of frames is easy to obtain. FRORL also allows the use of some keywords of structural programming, like "while do", "parbegin", and "parend" in order to enhance the specification readability and the expressiveness. Please refer to [Tsai93] for the syntax of FRORL language.

3. Verification Method

There are various properties of a requirements specification, like consistency, liveness, safeness, correctness, etc., of interest to software engineers. Once a specification has been constructed using the FRORL language, it may be subject to analysis tools which attempt to determine whether such properties hold for a given application problem. In this checking process, the user, with the aid of the system, goes through a sequence of iterations of modification and verification until the specification is assumed to be correct.

FRORL employs various strategies when analyzing a specification, namely (i) resolution refutation (for reachability, reversibility, liveness, consistency, synchronic distance, and bounded fairness), (ii) model checking in temporal logic, (iii) graph-theoretical algorithms for the determination of the consistency of timing constraints.

Note that analysis is performed at the level of the underlying logic, rather than at the level of the FRORL surface syntax of frames and rules. Throughout this section, we assume we are dealing with the specification as translated into a theory of the underlying logic. We speak, for example, of clauses of a FRORL specification, by which we mean "clauses of the theory obtained by translating the FRORL language into the base logic."

3.1. Analysis through Resolution Refutation

As far as logic-based reasoning is concerned, to analyze a specification for dynamic properties (i.e., functional properties that arise from the execution of the specification) one can rely on either a state-space strategy or a problem-reduction strategy. The state-space strategy is based on data-driven/bottom-up forward reasoning. This reasoning employs two sets of entities. The first set is a collection of *states*, where each state reflects the condition/status of the problem at each stage on the way to its solution. The other is a collection of *operators* which help to transform the problem from one state to another. The problem-reduction strategy is based on a goal-directed/top-down backward reasoning, which again involves two sets of entities: A collection of *goals/subgoals* which describe the problem, and a collection of *operators* which convert a goal/subgoal into more refined, or detailed, conjunctive subgoals. It can be shown [Murata88] that the problem-reduction strategy of a problem is logically equivalent to a state-space strategy of the same problem. The development of a FRORL specification is based on object-oriented top-down design methodology, and hence, FRORL essentially relies on a problem-reduction strategy. In modeling a system composed of a set processes, FRORL creates a conjunctive goal (a top level activity), with each goal modeling a specific process in the system. The state of a system process is assumed to be the sum of the values of its goal arguments.

Each process is decomposed into several subprocesses, and likewise, each goal is refined into many subgoals(lower-level activities). The state of a system is thought to be the union of the states of its components processes, which corresponds to conjunction of its goal arguments. The sole operator employed is the resolution rule.

All the dynamic properties of the specification discussed in this subsection will be verified through the construction of resolution refutations from the theory derived from the specification and a chosen goal clause.

- *Reversibility* - Determines whether an *initial goal* of a specification can always be resumed. Again, we extend this concept to arbitrary goals, to confirm whether a *specific goal* can be resumed.

- *Liveness* - Ensures that every goal in a specification is resumable from any other reachable goal.

- *Consistency* - Consistency analysis determines whether contradictions occur between the clauses of a FRORL-based specification.

- *Synchronic Distance* - In testing and debugging a specification, it is helpful if we know the mutual dependence among rules. *Synchronic distance* measures the correlation between two rules, i.e., their relevancy to one another.

- *Bounded Fairness* - Two clauses R_i and R_j of a FRORL specification are in a *bounded-fair* relation, if there is a bound on the number of times one is invoked while the other is not.

3.2. Analysis through Model Checking

In [Tsai93], we presented a temporal logic, RTμ, to model the time-dependent aspects of a specification. We also gave a method for determining whether a structure is a model for a given sentence of the temporal calculus, i.e., whether a given sentence is true in a structure. To verify a program through model checking treat the specification as a structure as described in [Tsai93]. Then determine whether this structure is a model for sentence of RTμ that expresses a desired property of the specification. The procedure for determining whether a structure is a model of RTμ has acceptable time-complexity. Model checking gives us a powerful mechanism to determine the correctness of our specification relative to a wide variety of temporal properties. In the following section, we discuss possible properties we propose to verify.

3.3. Temporal Properties

Sentences of the temporal calculus RTμ express properties of computation sequences. If a given specification is a model for such a sentence Φ of RTμ, we say that the specification has the property Φ. We will develop algorithms to determine whether a specification is indeed a model for a sentence of RTμ. In this section, we discuss properties of particular interest.

Safety properties intuitively assert that "nothing bad happens" during the computation. Less colloquially, a safety property states that a finite prefix of a computation(the computation may well be infinite) satisfies some condition. Typically safety properties have the form $\Box\phi$, where ϕ specifies the condition that has to be met within the finite prefix of the computation, and the modal operator $\Box$ ensures that ϕ holds of all finite prefixes. The following are examples of safety properties.

- *Partial correctness* - We say that a specification is partially correct with respect to a precondition ϕ and a postcondition ψ, provided that it starts to execute in a state satisfying the precondition, it will terminate in a state satisfying the postcondition, should it terminate.

- *Invariance* - That a property ϕ holds throughout the reasoning of the specification.

- *Deadlock freedom* - A set of processes is said to be deadlocked, if no process is enabled to proceed.

Roughly speaking, *liveness* properties assert that "something good will eventually happen." They require that for some element of a finite prefix of a computation some condition holds.

- *Total correctness* - A specification is said to be totally correct with respect to its precondition ϕ and postcondition ψ if it is partially correct and the program terminates.
- *Guaranteed accessibility* - for a process states that if a process reaches a certain state in the computation, it will eventually reach some other state. This property is sometimes referred to as absence of starvation. (Starvation freedom also refers to properties such as guaranteed of requested resources, etc.).

Intuitively speaking, *fairness* gives every process the chance to make some progress in its computation, independent of the relative speeds of the individual processes. Various fairness properties of different strengths have been proposed. Among them are:

- *Unconditional fairness* - Every process is executed infinitely.
- *Weak fairness* - Every process that is enabled almost everywhere is executed infinitely often.
- *Strong fairness* - Every process enabled infinitely often is executed infinitely often.

The properties listed above are templates of how one would express fairness, mutual exclusion, etc. for a concurrent language. These must be adjusted to the individual case.

3.4. Timing Constraints Consistency Analysis

The timing constraints of a real time system can be classified into performance constraints and behavior constraints [Tsai94b]. Performance constraints set the response-time limits on the controlling subsystem whereas behavior constraints make demands on the rates at which the controlled subsystems provides stimuli to the controlling subsystem.

The correctness of a real time system depends not only on the logical result of the computation, but also on the time at which the results are produced [Tsai90]. An important aspect of the correctness of a real time system is the consistency of the various timing constraints imposed on the individual component of a system. The system will not function correctly if even two of the timing constraints cannot jointly be met.

The timing constraints consistency analysis includes the following steps:

Step 1: Generate directed timing constraints graph (DTCG) and the corresponding directed timing constraints matrix (DTCM)

Based on the timing related parts of a FRORL language, we create a DTCG by letting the vertices be the events mentioned in the specification, and the arrows (with their associated weights) be the timing constraints that hold between two events. The arrows point in the direction of the timing constraint (e.g., for a maximum timing constraint between event e_1 and event e_2 the arrow goes from e_1 to e_2, and vice versa for a minimum timing constraint)

Step 2: Transform DTCM to canonical form

We shall locate all the cycles in a DTCG and compute the timing constraints on the cycles. First, we eliminate all events and arrows from the DTCG which cannot contribute to a cycle, i.e., all events (and their connected arrows) from which either no arrow originates or at which no arrow ends.

Step 3: Locate timing constraints contradictions

Now find cycles with the sum of the weight of the contained arrows being positive, and compute the overall timing constraints values.

4. Debugging of Logic-Based Specifications

FRORL is a formal specification language, which is capable of describing non-deterministic and non-monotonic nature of a software [Tsai92a]. We have made use of the concept developed by Shapiro [Shapi87] to perform debugging on FRORL specifications. This study aids the user in detecting and correcting the possible bugs which arise when writing the specification or even after the verification of the specification for the hidden bugs. The following sections show the algorithms applied to FRORL and brief analysis of complexities. Because of the nature of the domain handled by FRORL in addition to the original algorithms, we have added some tailored algorithms for FRORL [Tsai94a].

The algorithms and terms are rephrased here in order to fit the FRORL terminology. A set of a triples in the form of (a, x, y) which consists of an activity a along with its input x and output y forms an interpretation $\mathcal{I}$. If (a, x, y) produces a sequence $\{(a_1, x_1, y_1), (a_2, x_2, y_2), \cdots\}$ which is a subset of $\mathcal{I}$, then an activity a is said to cover (a, x, y) with reference to $\mathcal{I}$.

4.1. Diagnosing Termination With Incorrect Output

The legal computation of a specification are described via a computation tree which has the specification $\mathcal{S}$ as the root and the triples (x, y, z)'s as child nodes. The computation trees of a sub-specification are a subset of the computation tree of a specification which is formed by the sub-specification. The most primitive algorithm is called the single-stepping algorithm, which makes use of the technique of single-step through the activity calls of the computation. For each clause, a query is generated for the user to answer yes or no.

A. Divide-and-Query Diagnosis Algorithm

This is an improved method over single-stepping query strategy by querying the node (b, u, v) in the computation tree. This will divide the tree into two approximately equal parts. If (b, u, v) is in $\mathcal{I}$, then omit the subtree rooted at the node and repeat dividing. Otherwise apply the algorithm recursively to that subtree. The length and depth are linear in length of the original computation when the algorithm is implemented.

B. Algorithm for Tracing Preconditions

An activity frame contains a precondition slot which determines whether the action slot is executed or the alt_action slot is executed. When an incorrect output is encountered, a backward trace can be performed. From the activity which is executed the last, the activity which has called it can be traced. The arguments passed are checked against the precondition of the new found activity, and a query is issued to the user to check the current state. This process is repeated until a suspicious activity is found.

C. Algorithm for Send/Receive

The process is similar to the algorithm for divide-and-query approach. However, for the distributed and real-time domain, the built-in functions **Send** and **Receive** used turn out to be good "leads" to the next activity which will be executed. Starting from the root of a computing tree, the activity which has a corresponding **Receive** to the **Send** in its parent activity is added as a child node. This process is repeated to create the sub-computational tree.

4.2. Diagnosing Finite Failures

A specification is said to be finitely fail on (a, x, y), if for every computation of a on x terminates and returns an output correct in $\mathcal{I}$, but no computation output y is returned. An activity a is said to be complete with respect to $\mathcal{I}$, if for any (a, x, y) in $\mathcal{I}$, a covers (a, x, y) with reference to $\mathcal{I}$. Otherwise a is said to be incomplete. The algorithm uses existential queries to detect such activities. An existential query is a pair (a, x), and the answer to an existential query (a, x) in an interpretation $\mathcal{I}$ is the set $\{y|(a, x, y)$ is in $\mathcal{I}\}$.

4.3. Diagnosing Nontermination

Well-founded-ordering (*wfo*) [Shapi87] on a non-empty set $\mathcal{O}$ is defined as a strict partial ordering on $\mathcal{O}$ that has no infinite descending sequences. A specification $\mathcal{S}$ is defined to be everywhere terminating iff there is a *wfo* in the set of activity calls such that every computation of $\mathcal{S}$ in which (a, x) calls (b, u). This is the case that $(a, x) \succ (b, u)$. An activity a is said to diverge with respect to and $\mathcal{I}$ if for (a, x, y),

1. there is a set (b, u, v) in $\mathcal{O}$ for which $(a, x) \not\succ (b, u)$, and
2. all triples in $\mathcal{O}$ that precede (b, u, v) are in $\mathcal{I}$.

An activity is said to be a loop if it calls itself with the same input it has been called with. Any activity that loops also diverges if activity calls that have preceded the looping call have returned a correct output. If $\mathcal{S}$ is diverging then it contains an activity incorrect in $\mathcal{I}$ or an activity that diverges with respect to $\mathcal{I}$. For every activity a and b in $\mathcal{S}$, the user is to answer queries of the form "is (a,x) $\succ$ (b,u)?". This leads to find a sequence, $\langle p, x\rangle \cdots \langle p, x\rangle$.

5. Modeling Non-Functional Requirements

Current requirements specifications tend to focus more on the functional aspect than the non-functional side of a product [Tsai94d]. There have been many techniques and tools developed to specify and evaluate functional requirements, but few of them are equipped to deal with non-functional requirements. In our research project, we extend the formal requirements specification language, FRORL, to model non-functional requirements and show how these non-functional requirements are related to the functional requirements. We also introduce a parallel evaluation technique to evaluate the functional requirements model by satisfying the non-functional requirements associated to it. By examining the result from the functional model, we can see how the non-functional requirements are satisfied, so that we can adjust and modify the non-functional requirements accordingly.

6. Issues in Non-Functional Requirements

When modeling non-functional requirements, we need to consider both *what* we model and *how* we process the model. The attributes of a system are *what* we model, and we can use either the process-oriented approach or the product-oriented approach to deal with *how* we process the model. As non-functional requirements are the quality issues in a software system for describing different attributes in a system. The attributes like *performance*, *accuracy*, *security*, or *user-friendliness* are the quality needed to be seen in a software system, but are not totally defined in functional requirements.

For different attributes, we have different types of models. One can be just the description of what an attribute is; another can be the relation of how an attribute is constructed by other attributes. These are modeled by the *product-oriented approach* and the *process-oriented approach* respectively. We can apply a qualitative approach which uses reasoning to see if the model is

correct or a quantitative approach which relies on software metrics to produce measurements on the model.

The *process-oriented approach* concerns on how well the non-functional requirements are modeled. It checks to see if a certain piece of requirements can be derived from the given conditions. The idea can be seen as a goal-driven reasoning, where a requirement is a goal to meet. The model can be incomplete, because reasoning on attributes can produce results. This process of reasoning is a qualitative approach for verifying the model. The *product-oriented approach* basically focuses on the outcome of the model as a whole. It is like measuring a final product with a software metric technique to see how well the performance criteria are met. The measurement is based on the quantitative approach, because the major concern is to obtain a number which can be analyzed based on some standards. Non-functional requirements are measured according to the criteria set by the analyst.

An important issue in modeling non-functional requirements is to identify the attributes which are significant to the application. This paper presents a process-oriented method for modeling non-functional requirements using FRORL language. The non-functional requirements are modeled as object frames in FRORL. The root frame (node) of an object tree (or a sub-tree) is satisfied when its child frames are satisfied. The abstract relation between objects have the following meaning:

an_instance_of binding values to variables.

a_kind_of OR relation among siblings.

a_part_of AND relation among siblings.

Non-functional requirements are related with each other tightly. Changing one requirement may affect another. For example, higher accuracy requirements may result in lower response time and lower system throughput. This relation is modeled as an activity frame. The parts in the activity frame contains the non-functional requirements involved in the relation, and the actions defines which objects are affected by other objects.

After we have a non-functional requirements built, we need to associate non-functional requirements to functional requirements. An object attribute, called *property*, is used for stating what kind of non-functional requirements is defined for that particular object. The non-functional requirements of a parent frame will be inherited by the child object frames. A child frame can also override the non-functional requirements specified in its antecedent frames by associating with a a different kind of non-functional requirements. This enables the non-functional requirements at root to represent some general and abstract specification and refine these definitions along the frame hierarchy.

The non-functional requirements of an activity are implicitly specified. An activity has a parts slot which contains object names. When the objects in the parts slot are associated to some non-functional requirements, then the activity will assume those requirements. For example, if an activity, deposit, has two parts, amount and account and if the object, account, has non-functional requirements of accurate, then the result of the activity, deposit, must be accurate.

7. Parallel Evaluation Mechanism

Many techniques have been proposed to execute a logic program in parallel. Current approaches seen in the literature adopt a top-down evaluation sequence to realize AND-OR parallelism by exploring executable predicates in parallel level by level from root to leaf and then collect bindings generated in this process. There are two major problems involve in this AND-OR parallel evaluation of a logic program. One is the binding conflict at AND-parallel node,

another is the synchronization at OR-parallel node. The former arises when different predicates in a clause body bind different values to a same variable while evaluated, the later occurs when values generated from different branches of an OR-parallel node have to be collected and merged into a complete answer set and passed onto a higher level. The synchronization problem has been solved by including various data structures at each OR-parallel level for collection purpose. This approach introduces large overhead on the system. The binding conflict has been solved primarily by introducing data dependency in the program so that only one producer can exist at any time. This approach may reduce the degree of parallelism during execution.

In order to handle AND-OR parallelism, a bottom-up combined with top-down evaluation scheme is used in our parallel evaluation model [Tsai94c]. The approach can find all the mode combination before the user's query is available, it is possible that while the execution goes on, some information about the predicates which is to be executed several steps later is made available in advance. If we carry out those operations first, then more mode information for un-searched predicates can be gathered in advance and the efficiency of the whole process can be improved. The approach uses the result generated from the static data flow analysis, combined with the user's input, to find out the portion of the AND-OR tree of which the execution behavior is deterministic under the current variable binding. This portion of the AND-OR tree is executed in a bottom-up fashion. Comparing with other approaches, which rely totally on the user's input query to initiate the analysis, their execution are totally based on the dynamic determination of data flow information and the dynamic analysis which inevitably puts tremendous overhead on the system.

The overhead for our model is the cost for the static mode analysis. This portion of the cost is the same among all the approaches which adopt mode information. The cost to finish a mode checking for a predicate is $O(n*m)$, where n is the average number of predicates within a clause, m is the number of arguments within a clause. Some extra overheads involving communication also occur in this evaluation model. A simulator has been implemented to analyze the execution behavior of the new model. Along with significant improvement obtained, detail discussion of this parallel evaluation mechanism can be found in [Tsai94c].

8. Parallel Analysis of Non-Functional Requirements

The evaluation of non-functional requirements can be carried out during the parallel execution [Tsai94d]. Results regarding the execution behavior and the functionality the system performs can be gathered and evaluated by the parallel evaluation system by referring to the non-functional requirements specification. This non-functional reference is realized by the objects which act as parts of the activity frame consisting the parallel evaluation. The result is then reported to the user to further adjust non-functional requirements specification. Because of the parallel evaluation model, the adjustment of non-functional requirements specification can be implemented with higher efficiency.

9. Logic-Based Transformation System

The transformation subsystem will perform the transformation from the developer's specification to various target codes such as C, Ada, Prolog, and others. The input to our transformation system is FRORL specifications, and the output is the target code written in a procedural programming language which has the same description power as that of the original FRORL specification. There are three main steps in our transformation system [Tsai92b]. The first step is the transformation from a FRORL specification into a Horn clause Logic-based specifications. The second step consists of pre-transformation, data flow analysis and execution sequence determination of the logic-based specification. The pre-transformation includes *Equal-Introduction*, *Equal-Substitution*, *Decomposition* and *Simplification*, which is used to transform a logic-based

specification into some entailed form but still within logic specification category in order to make the data flow and execution sequence analysis easier to apply. These operations often need to be applied again and again to make necessary adjustment to the pre-transformation of a logic-based specification along with the progression of transformation process. The data flow analysis and execution sequence determination is used to automatically deduce mode information of clauses in a logic-based specification without the user's input. These mode information is essential to make adjustment to the execution sequence of a logic-based specification so that the execution sequence of the adjusted specification can be represented by a procedural language program which is normally sequential. The last part of our transformation system is modeling backtracking control mechanism and target code generation. We have derived rules to represent these transformation knowledge in [Tsai93].

10. Conclusion and Future Research

This paper presents a framework for a specification-based software architecture using a frame-and-rule oriented requirements specifications language. The language, FRORL, serves as multiple roles for specifying users' requirements, for representing development knowledge, and for knowledge communications among various software tools. Various verification and debugging methods have also been developed on the logic-based specification. Using a new parallel evaluation model, validation and verification process can be speeded up. Overall, the system provides an uniform framework to rapid software development with the specifications prototyping and the transformation into target programming languages.

Currently, the language construct of FRORL and the supporting facilities did not be fully implemented in real-time distributed environments though the language does have concurrent features. This will be the future goal to provide prototyping and analysis tools for real-time distributed computing and multimedia applications. It requires the extended semantics of FRORL and its corresponding theoretical foundations in order to specify and analyze various information in an autonomous distributed environment.

Acknowledgement

This research is supported in part by Fujitsu Network of Switching, Inc.

References

[Garlan95] D. Garlan and D. Perry, "Introduction to the Special Issue on Software Architecture," *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 269-274, April 1995.

[Luqi95] Luqi and V. Berzins, "Software Architectures in Computer-Aided Prototyping," *Proc. 1995 Monterey Workshop*, Sept. 1995.

[Murata88] T. Murata and D. Zhang, "A Predicate-Transition Net Model for Parallel Interpretation of Logic Programs," *IEEE Transactions on Software Engineering*, Vol. 14, No. 4, pp. 481-497, April 1988.

[Shapi87] E. Shapiro, *Concurrent Prolog,* MIT Press, Cambridge, MA, 1987.

[Tsai88] J. J.P. Tsai, "A Knowledge-Based Approach to Software Design," *IEEE Journal on Selected Areas in Communication,* Vol. 6, No. 5, pp. 828-841, June 1988.

[Tsai90] J. J.P. Tsai, K. Y. Fang, H. Y. Chen, and Y. Bi, "A Non-Interference Monitoring and Replay Mechanism for Real- Time Software Testing and Debugging," *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 8, pp. 897-916, August 1990.

[Tsai92a] J. J.P. Tsai, T. Weigert, and H. C. Jang, "A Hybrid Knowledge Representation as a Basis of Requirement Specification and Specification Analysis," *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, pp. 1076-1100, December 1992.

[Tsai92b] J. J.P. Tsai, R. Sheu, and B. Li, "A Framework of Logic-Based Transformation Systems," *Proc. of IEEE 16th International Computer Software and Applications Conference*, September 1992, pp. 294-299.

[Tsai93] J. J.P. Tsai and T. Weigert, *Knowledge-Based Software Development for Real-Time Distributed Systems*, World Scientific, 1993.

[Wei94] T. Weigert and J. J.P. Tsai, "A Computationally Tractable Non-Monotonic Logic," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 1, pp. 57-63, February 1994.

[Tsai94a] J. J.P. Tsai, A. Liu, and K. Nair, "Debugging Logic-Based Requirements Specifications for Safety-Critical Systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 2, pp. 205–228, June 1994.

[Tsai94b] J. J.P. Tsai and S. J.H. Yang, *Monitoring and Debugging Distributed and/or Real-Time Systems*, IEEE Computer Society Press, Washington D.C., 1994.

[Tsai94c] J. J.P. Tsai and B. Li, "Improving Parallel Execution Performance for Logic Programs Using Mode Information," in *Proc. of IEEE 6th Parallel and Distributed Processing Symposium*, Dallas, TX, October 1994.

[Tsai94d] J. J.P. Tsai, B. Li, and A. Liu, "Modelling and Parallel Evaluation of Non-Functional Requirements Using FRORL Language," in *Proc. of IEEE 17th International Computer Software and Applications Conference*, November 1994.

Software Architectures in Computer-Aided Prototyping *

Luqi, Valdis Berzins
Computer Science Department
Naval Postgraduate School, Monterey, CA 93943, USA

Abstract

We examine representations and support for software architectures in the context of computer aided prototyping. To assess the potential contributions of advances in this area, we explore the connection between generic software architectures and automation support for software reuse, program generation, software evolution, reengineering, and transformation of prototypes into production code.

1 Introduction

The study of software architecture is concerned with the large scale structure and design of software systems. A software architecture defines the common structure of a family of systems by identifying and specifying (1) the components that comprise systems in the family, (2) the relationships and interactions between the components, and (3) the rationale for the design decisions embodied in this information. The concept of software architecture was generalized to cover the structure of a family of systems rather than just a single system to better support software evolution and reuse. Software architecture is a relatively new field of study, and many variations on this basic idea have been proposed [1].

Languages are being developed to describe software architectures. An architecture language has to be based on a model of software architectures, and the properties of that model have a strong influence on the complexity and usefulness of the language. Designing a general purpose architecture language is very difficult because there is such a wide range of architectural possibilities, especially if all details are included. This makes it very hard to cover the subject without making the language so complicated that it becomes difficult to use. The problem is somewhat easier if the application domain is narrowed to address only one type of application. For example, the types of architectures used in business information systems are different from those used in embedded real-time systems.

This paper examines software architectures in the context of computer-aided prototyping, explains the architecture models used in that context, presents some examples of representations

*This research was supported in part by the National Science Foundation under grant number CCR-9058453 and in part by the Army Research Office under grant number ARO 111-95.

for software architectures, and outlines some of the associated automation support. We also indicate some extensions to these representations and relate our results to software architectures in a more general context.

2 Related Work

Software architectures are receiving increasing attention because they are closely related to many aspects of software development:

Synthesis. Large and complex systems are realized by interconnecting components in hierarchical assemblies. Thus support for designing interconnections of subsystems is a key to rapid and economical construction of large systems [8].

Understanding. People can understand complex designs only by organizing them into levels that have relatively small and simple realizations in terms of the components at the next level down. Since understanding is a prerequisite for quality control, the choice of software architecture strongly influences the reliability and appropriateness of software products.

Evolution. It is necessary to understand the structure of a system and its principles of operation in order to plan and reliably adapt the design to a requirements change. Since the structure of the system has a strong influence on the cost of a change, the well known principle of information hiding [10] urges system architects to confine decisions that are likely to change within individual components. It is quite difficult to anticipate what decisions will change, so that the right software architecture is often found only after several attempts at designing systems in a given domain. Improvements in support for designing and modifying software architectures can increase software flexibility [5].

Reuse. The reusability of both components and interconnection structures is critically dependent on architectural coherence. Experience has shown that prior planning and deliberate design for multiple use are needed for effective reuse. Arbitrary pieces of code are usually not reusable. Reusable components are designed to fit into architectures that cover a large span of applications, and reusable architectures are consistent with many well-defined options for component behavior.

Integration. Different systems cannot work together effectively unless their interfaces are consistent. Software architectures define the standards, conventions, and common conceptual models needed to achieve such consistency [3]. Thus agreement on a common architecture at the highest levels is a prerequisite for interoperability of systems developed by different organizations.

Analysis. Explicit representations of architectural information enable new forms of computer aided software analysis and decision support for system designers, such as consistency checking [3] and real-time scheduling [8]. The localized structure that enables human understanding also appears to be needed for automated decision support, because the computational requirements for many analysis tasks increase sharply with the number of components to be considered.

Management. The structure of the software architecture is closely related to the structure of the human activities needed to construct or modify the software system. Explicit representations for software architectures thus enable decision support for project planning and completely automated scheduling, work assignments, and configuration management [2]. The design rationale aspect of software architectures enables decision support for planning software evolution efforts [11].

Current best commercial practice with respect to software architectures appears to center on toolkits and frameworks with limited automated decision support [9]. Experimental languages for describing software architectures and composing systems from large-scale components (megaprogramming) have been proposed.

The megaprogramming language MPL considers the problem from a database perspective, and considers issues such as data transformations required when crossing subsystem boundaries, optimization of large scale actions, and dynamic monitoring of progress to control scheduling and execution strategies [13]. The architecture language UniCon specifies both software functionality as well as packaging properties of software components and connectors [1]. The system architecture language Rapide is aimed at distributed systems, captures dynamic as well as static connection patterns, and provides simulation and behavior analysis functions [1]. These languages address complex software products, and strive for comprehensive coverage of architectural properties. The languages themselves are also fairly complicated.

Experimental systems are also emerging for automatically composing programs for solving problems over a fixed problem domain based on a given architecture and a set of components consistent with that architecture. These systems take a somewhat different approach: the architectural information is not intended for human consumption, but rather is used internally by software tools that automatically create instances of the architecture that realize particular applications. Some of the systems in this category include AMPHION, a system for constructing programs that do astronomical calculations from graphical descriptions of the situations to be analyzed [7], Panel, a system for constructing multimedia animations [7], SDDR, a system for creating reliable and reusable software designs [7], ControlH and MetaH, systems for developing control software [7], and CAPS, a system for creating prototypes of real-time systems [8].

Our interest in software architectures is motivated by the desire to provide computer aid for the software prototyping process [6]. Iterative prototyping is characterized by repeated and substantial changes that focus on a common theme determined by the known and unknown needs of the clients and the areas of the greatest uncertainty. The requirements and a software architecture for realizing those requirements are developed concurrently via an iterative process that uses prototype demonstration to elicit adjustments to requirements. The software architecture is developed as a necessary by product, which is required to realize the executable version of the prototype. The software architecture serves a dual role in this process, because it serves both as the design for the initial version and as a description of a family of similar systems. Each step in the prototyping process can be viewed as navigation in this family of systems, with the goal of moving from the current point to one closer to the needs of the clients.

The focus of our work has been to support rapid prototyping and large scale software design, particularly for large, distributed, and real-time systems. The prototyping language PSDL associated with the CAPS system is an early architecture language tailored to support automated

generation of connections, automated real-time scheduling, computer-aided software reuse, and computer-aided software evolution [8]. PSDL is coupled with a variety of formal notations for describing behavior of individual software components, such as the specification language Spec [3].

CAPS and PSDL were developed before software architectures emerged as an independent subject. Although the purpose of the CAPS project was to provide computer aid for prototyping rather than developing deliverable software, the effort addressed many problems related to software architectures. The rest of this paper summarizes these results and suggests some extensions that contribute to software architectures in a wider context.

3 Representing Architectures in Prototyping

Prototyping requires rapid realization and analysis of proposed system behaviors, so that designers can iteratively approach an accurate formulation of client needs and corresponding software solutions. PSDL was designed to achieve the required speed through *simplicity.*

This goal was achieved by introducing architectural abstractions to eliminate as many details from the designer's consideration as possible, to separate the specification of the essential aspects of the abstract architecture from implementation aspects of the concrete architecture, and to minimize the implementation aspects imposed on the designer by automating the choice of as many implementation details as possible.

The result is a spartan representation for abstract system architectures, summarized in Fig. 1. A PSDL architecture is a connection pattern that specifies how a set of components are to interact by defining connections and constraints. The rationale for the structure is represented via links to system requirements together with formal and/or informal descriptions of intended component behavior. This information is organized in a hierarchy. Generalization is supported by generic components and connections. The rest of this section describes these aspects of the architecture model underlying the PSDL representation in more detail.

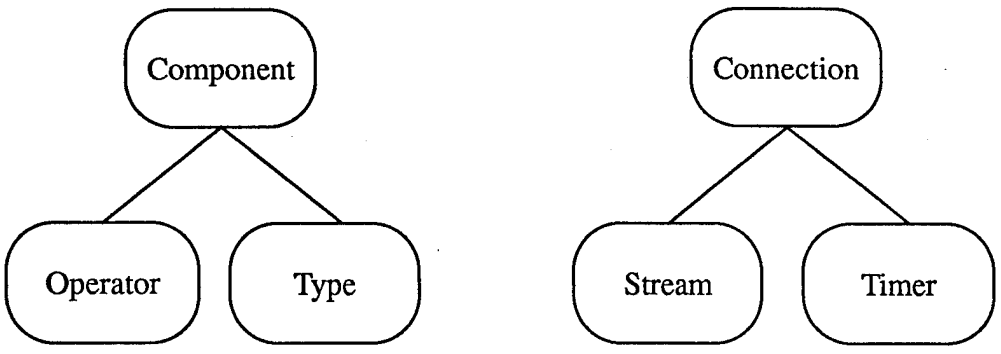


Figure 1: PSDL Component and Connection Types.

3.1 Interactions

PSDL has just two generalized types of interactions between systems: data streams and timers. This model is the simplest possible in the sense that these types of interactions are orthogonal: the first is mediated by transmission of data and the second is mediated by the passage of time. Experience has shown that they are sufficient for describing a variety of embedded real-time systems.

PSDL models connection patterns for operators as annotated networks of subordinate operators communicating via data streams. This model can be represented mathematically as an augmented directed hypergraph whose nodes are operators and whose edges are streams. Edges in a directed hypergraph can have multiple sources (operators writing into the stream) and multiple sinks (operators reading from the stream). This structure can be displayed graphically in a format similar to traditional data flow diagrams.

A simple computational model is associated with this structure. When an operator fires, it reads one data value from each of its input and state streams, and writes at most one data value into each of its output and state streams.

The hypergraph representing the connection pattern is annotated with timing and control constraints attached to the operators and streams. The timing and control constraints determine the conditions under which the operators are activated (i.e. can be fired).

The main simplification realized in the component interaction model is due to a general abstract model of system interactions that hides details of protocols and unifies data and control flow into a single type of interaction, the data stream. Data streams are generalized objects that have specializations with respect to several dimensions, including the data type whose values are carried by the stream, whether the stream represents a state variable, whether the stream models a discrete or continuous data source, whether the stream connects different processors, and the implementation languages of the producer and consumer systems. The data type carried by a stream is part of the abstract architecture of the system and must be specified explicitly by the designer.

The other properties are hidden from view because they are automatically derived from the abstract and implementation properties of the subsystems that interact via the stream. This simplifies the designer's view and removes an opportunity for introducing inconsistencies into the architecture. This represents a radical departure from other approaches to architecture. Rather than treating a component as an individual and specific piece of code that has a specific and detailed communication protocol, we consider the component as a module with abstract input and output patterns that can be realized in a variety of concrete protocols as the need arises, via generated code. At a high level of design, packaging aspects are irrelevant. If connections can be automatically realized, then packaging aspects become largely irrelevant for concrete realizations as well, at least in the context of prototyping.

Concrete packaging can impact optimizations that may have to be performed to transform prototypes into product-quality implementations. High level representations for this kind of information have been explored in the context of the Spec language, where they are represented via optional refinement declarations that define implementation strategies [3]. This structure is a high level analog to pragmas in Ada, except that the declarations represent binding constraints rather than optional implementation advice. Treating component interactions abstractly and providing

automated support for realizing and choosing concrete packaging aspects of component interactions is attractive for production code as well as for prototyping because it eases performance tuning of large systems.

Communication via side effects on shared mutable objects is excluded from the abstract architecture in the context of prototyping because such constructions can easily introduce faults due to unplanned interactions. Preventing this failure mode speeds up prototype realization and evolution by reducing debugging time.

This prohibition does not exclude concrete implementations that communicate via side effects. It does confine these mechanisms to optimizations introduced in the automatically generated realizations of streams. Such optimizations are strictly implementation level details that have no effect on the abstract architecture: the connection generator has an obligation to ensure that the behavior of optimized realizations conforms exactly to the behavior of the abstract architecture. This supports one of the central principles of large scale software design: subsystems must interact *only* via the specified interfaces.

The connection model of PSDL also supports another basic principle of large scale software design, which says that the specification and implementation of a component should be independent of the context in which it is used. The interface to an operation refers only to the input streams and output streams of the operation. Both the specification and the implementation of each operation are completely independent of where the data in the input streams comes from and where the data in the output streams goes. Thus an operation can be connected into compatible slots in many different architectures without any fear that this might inadvertently change the behavior of the component in some way. This enhances reusability and flexibility of both components and connection patterns.

PSDL uses a variety of notations for defining the required behavior of components, including Spec. Similarly to PSDL, Spec was designed to provide a unifying abstraction for interactions between subsystems. The spec model of interactions is based on the event model [3], in which data items are associated with events caused by data transmissions. This abstracts from differences in realizations of control and data connections, such as subprogram calls vs. rendezvous vs. gotos and parameter passing vs. I/O vs. global data. The Spec model of interactions is more restricted than the PSDL model. In Spec all of the data components associated with an event come from the same source, while in PSDL the input streams of an operator can come from different sources. This difference is not significant when both notations are used together because Spec can be used within PSDL only to define the behavioral requirements for an individual component slot in the architecture.

3.2 Components

The PSDL component model is also very simple: all components are either data types or operators (logical processes). Abstract architectural aspects of components are separated from incidental aspects of concrete realizations. Implementation considerations such as packaging into physical processes, mutual exclusion, locking, synchronization, and flow of control are all automatically realized by generated code, based on declared control and timing constraints associated with the operators.

At the logical level, all data types occurring in a PSDL architecture are immutable. At the

level of concrete realizations, mutable representations are allowed as optimizations provided that they correspond exactly to the specified abstract behavior.

All states in the abstract architecture are associated with data streams or timers representing state variables. Every state variable is local to some operator component, although its current value can be transmitted on output streams. This realizes the strict encapsulation required for independent modules.

3.3 Constraints

PSDL connection graphs are augmented with constraints that are associated with component slots in the architecture. These constraints serve several purposes:

- Some control constraints define the conditions under which each component can be activated. These act as execution guards and document the assumptions that can be made in the implementation of the component that fits into the architecture slot.
- Other control constraints limit or augment the runtime behavior of the components. These constraints serve to make small adjustments to the behavior of existing components. This capability enables reuse of components that do not quite match the designer's needs and supports evolution of systems containing legacy code whose source is not available for modification. Examples are output guards that suppress component outputs that do not satisfy specified conditions, exception constraints that raise exceptions if outputs do not satisfy specified conditions, and timer constraints that can start, stop, and reset timers under specified conditions.
- Timing constraints specify how often and how quickly operations must be performed. These constraints identify the time-critical aspects of the system and determine how computing resources must be allocated to meet the timing requirements associated with interactions between components.

The control constraints are realized by automatically generated code that realizes the interactions between the subsystems. The timing constraints are realized by automatically generated code that embodies schedule and resource allocation decisions made by the supporting tools.

3.4 Rationale

There are two types of rationale information associated with a PSDL architecture: component specifications and requirements links.

Component specifications describe the required behavior for the components that can fit into a slot in the architecture. These specifications can be used as queries against a software base to automatically search for reusable components that can fill the slot. The specifications also serve to document the requirements of the slot as well as the degrees of freedom that the architecture allows for filling the slot. These requirements identify the properties of a component that should be tested or proved to ensure that it can reliably satisfy the requirements of a given slot in the architecture. The Spec language provides a formal notation for component specifications that

can express partial constraints on component behavior and generic parameterized specifications as well as completely determined specifications for individual components.

Requirements links connect parts of the architecture to higher level requirements that motivated the parts. All aspects of the architecture description can have requirements links, including individual components, connections, and constraints. The requirements are named entities that represent goals of the stakeholders of the system. They are represented informally, in terms that the stakeholders of the system understand. The requirements links support user reviews of an architecture as well as tools for supporting the evolution of the architecture.

3.5 Hierarchy

In PSDL, a software architecture is a hierarchical structure that can be represented as an operator realization tree and a type realization graph.

The root of the operator realization tree represents the entire system (the software and all external systems that interact with the software). Each operator in the tree is labeled with a connection pattern that defines its architecture, as illustrated in Fig. 2. The children of each operator in the tree are the component operators that appear in its connection pattern. The atomic operators at the leaves of the tree have empty connection patterns.

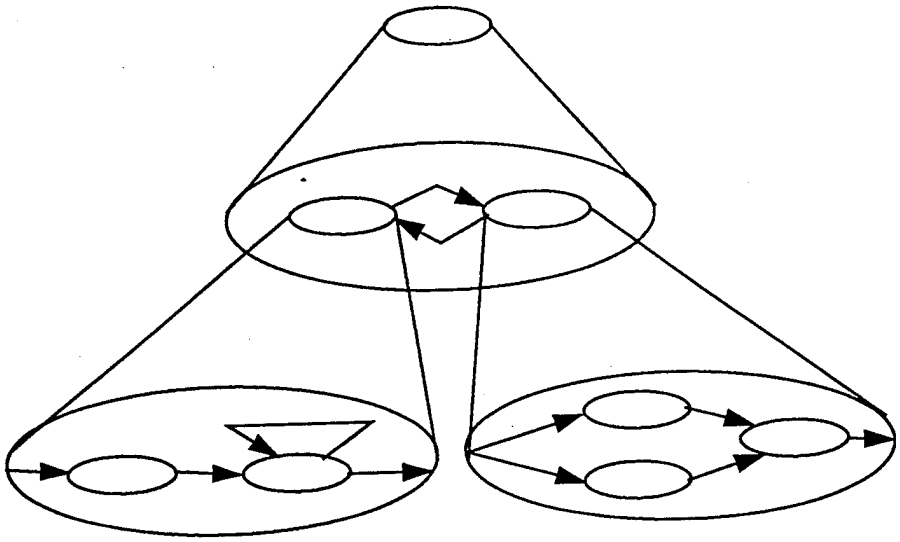


Figure 2: A Hierarchical Architecture.

Implicit in the tree representation for operator realizations is the constraint that each operator can appear in the realization of at most one parent operator. This is not a serious constraint on the designer and does not preclude reuse within a system because distinct copies of an operator can appear in several different places in the operator realization tree.

The prohibition against sharing affects the behavior of operators with internal states, because distinct copies of a state machine have distinct and independent copies of the state. Thus state transitions of one copy do not affect the states of all the other copies of the machine. This

constraint helps to ensure that all interactions between subsystems are apparent and explicitly specified in their interfaces.

The type realization graph contains a node for every data type associated with a stream in the connection patterns. The children of each type in the graph are the other types used in its representation. Atomic types are terminal nodes in the graph. Each type is an abstract data type, and is associated with a set of operators that act on the instances of the type. Each of these operators is the root of its own operator realization tree.

Atomic components, both operators and types, are realized by program modules that conform to the component slots in the architecture.

3.6 Example

Architectures are most useful if they are easy to tailor with respect to some dimensions, while still providing some common properties along other dimensions. A partial description of a simple generic architecture for a secure communications channel is shown in Fig. 3.

This generic architecture can be tailored to provide different security properties by plugging in different versions of the encoder and decoder components. However, these two components are not independent. One of the consistency properties associated with the architecture is shown in Fig. 4. This constraint requires that the message will be transmitted without modification for properly matched keys, for all well-formed realizations of the architecture.

The detailed security properties of the communications channel are not completely determined by the architecture, and can vary with the choice of the encoder and decoder components. For example, a simple realization might require that two keys form a matched pair if and only if they are equal. In this case both keys must clearly be kept secret. A more sophisticated realization with public encryption keys might require that the encryption key be the product of two large prime numbers, and the matching private decryption key be one of the prime factors of the public key. A version with a higher level of security might require the relation between matched pairs of keys to depend on the time the message was sent.

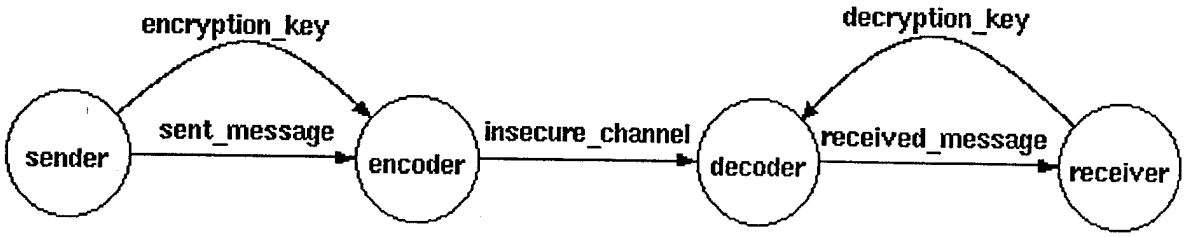


Figure 3: A Generic Secure Channel Architecture.

4 Automation support for software architectures

With respect to our focus on computer-aided prototyping, the most basic kind of support for software architectures is generating code for realizing the specified interactions between the com-

```
OPERATOR encoder
  SPECIFICATION
    INPUT
      encryption_key : key,
      sent_message : message
    OUTPUT
      insecure_channel : message
  AXIOMS
    { ALL(m: message, k1 k2: key SUCH THAT matched-pair(k1, k2) ::
          decoder(k1, encoder(k2, m)) = m) }
END
```

Figure 4: An Instantiation Constraint of the Secure Channel Architecture.

ponents. This was one of the first capabilities developed for the CAPS system [4].

Another basic kind of automation support is using the architecture description to find components that can fill slots in the architecture. These components can be realized by individual modules of software or hardware, or by connection patterns given by a lower level architecture. We have developed a method that uses practical amounts of computational resources and can in some cases certify that an automatically retrieved component will meet the user's requirements, so that it can be used without inspecting the code [12]. This method requires component specifications to be associated with the reusable components in the software base as well as with the component slots in the software architecture. There is a tradeoff between computational effort and the proportion of behavioral matches that can be detected by such a system. Completeness is unattainable because exact specification matching is undecidable for specifications with sufficient expressive power to represent the full range of designer intentions. Work is currently under way to explore related methods and representations for component behavior that are easier to use, to make this capability accessible to a larger group of software developers.

More recent efforts use architectural information to provide decision support for software evolution. This part of the effort uses a graph model to represent the derivation history of the architecture [5]. The graph contains different versions of the architectural information for the system, including hierarchies of connection patterns for the subsystems and requirements. Dependencies between the components are captured by representations of design steps that are linked to the versions of the architectural description units they produce and to the versions of other architectural description units that were used to derive the new version. This structure can represent parallel and reconverging threads of development as well as the more common linear chains of development steps. The structure contains proposed and planned steps in addition to completed steps and their products. Several kinds of support for software evolution and team coordination are based on this information.

The evolution control system of CAPS [2] uses the requirements dependency information together with the structure of the previous version of the architecture to derive an approximate

work breakdown structure for responding to a proposed requirements change. The project manager approves proposed changes, and adjusts the approximate work plans to account for possible new subsystems required by the change and to identify modules that do not change even though they contribute to the changed requirements. The manager also adds effort estimates and policy information such as priorities of different changes, deadlines, and skill requirements for different designer tasks. The system then uses information about the team of available designers to project a schedule, assign tasks to designers when they become free, and automatically monitor the progress of the project against the deadlines. It also automates configuration management based on the work plan by automatically checking out the proper versions of the documents needed to complete a design task, putting them into the responsible designer's private work space, and automatically checking the results back into the repository with the proper dependencies and version identifiers when the task is done and associated checking procedures have succeeded.

CAPS also has facilities for automatically combining the effects of two changes to an architecture, and for checking the semantic consistency of the two changes [8]. This facility is particularly useful when different aspects of an architecture are undergoing concurrent exploratory development.

5 Extensions

PSDL has a relatively static view of software architectures because it supports automatic methods for realizing hard real-time constraints on system behavior. The number of time-critical functions must be bounded to guarantee timing constraints can be met with fixed computational resources. This is achieved in PSDL by requiring that the set of components be statically declared. This implies that dynamic reconfiguration of software architectures is limited in the PSDL model: all possible components and connections must be statically declared, and resources must be allocated for the worst case. Reconfiguration in this context amounts to dynamically enabling or disabling selected connections and components from the fixed set of possibilities. Such reconfiguration is readily expressible via PSDL control constraints and a set of data streams carrying descriptions of the current system configuration.

This situation can be extended in several ways.

- Dynamic component creation can be allowed for components and connections that do not have any timing constraints. In such a case system response times can increase with the number of currently active instances of components in a dynamic software architecture.

- Dynamic component creation may be tractable for time-critical components if component creation is linked to creation of additional processors and connections. This model makes sense for large distributed systems with long lifetimes, where new hardware nodes can be added while the system is in operation. For example, it is entirely reasonable to assume that each new airplane will have its own set of onboard computers. However, limitations on the communications bandwidth and network diameter (maximum path length) still seem to require bounds on the maximum size of the dynamic configuration of such an architecture to guarantee service within a fixed deadline.

A challenge that arises in the description of dynamic architectures is developing understandable representations of the connection rules. For static systems a connection graph presents a convenient and understandable representation of component interactions that can be displayed and edited graphically. For dynamic systems more abstract representations are needed to capture the parts of the interaction protocols that remain invariant across the dynamic reconfigurations. Fully general solutions appear to require powerful notations that can be hard to read and can require high skill levels to use.

Readily understandable representations are possible if we constrain the degrees of freedom in the dynamic architecture. A simple example of a tractable and constrained kind of dynamic architecture is one that allows a variable number of instances of the same generic component that share a common role in a given interconnection pattern. This construction can be used for time-critical operations if the number of instances in the collection is bounded. An example of a possible graphical representation for such a connection pattern is shown in Fig. 5. The example is a fragment of an architecture for a display system that supports multiple windows.

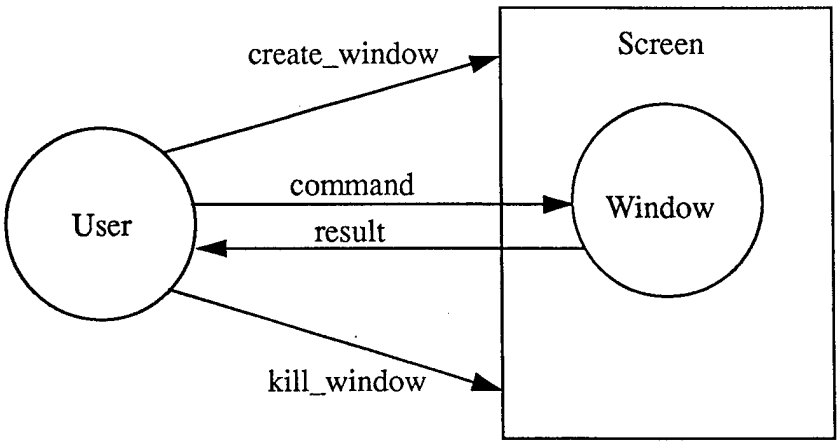


Figure 5: A Dynamic Architecture.

This example illustrates an extension to PSDL that supports dynamic collections of components as architectural building blocks. The component shown inside the collection box is a generic template that can have zero or more instances. In the example, the screen is a collection of windows. Creation of new windows and destruction of windows are operations performed by the collection; the streams create_window and kill_window are therefore directed to the collection rather than to the instances. These streams carry values of type window_id, which correspond to the generic parameter of the window template and serve to identify the instances of the collection to be added or removed.

The default communications pattern into such a collection is broadcast to all of the instances. The use of generic parameters to distinguish among the instances of the replicated component enables PSDL control constraints to specify more selective message routing, to single out either individual instances or larger subsets of the collection.

The default communications pattern out of such a collection is writing into a common stream, which implicitly merges all of the communications into a linearly ordered sequence based on

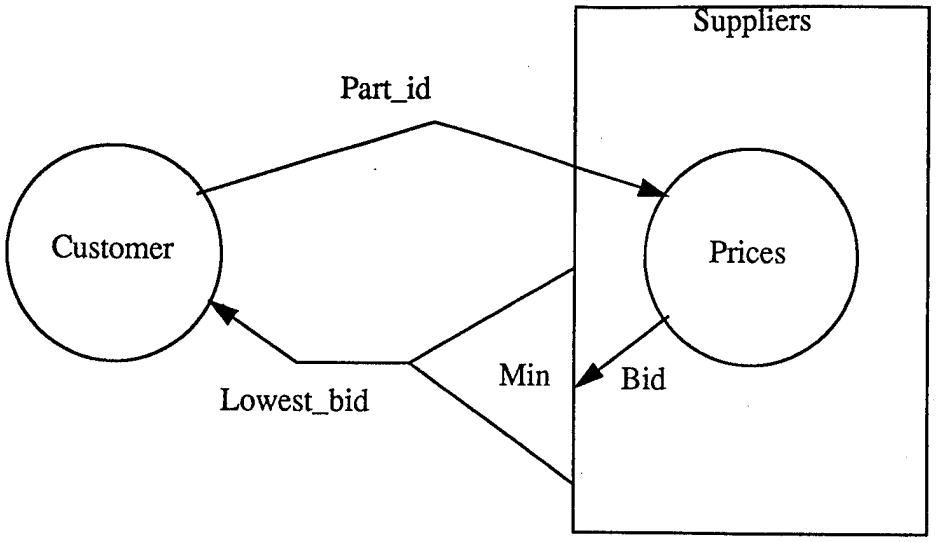


Figure 6: Reductions in A Dynamic Architecture.

writing time. The most common alternative to the default output pattern for collections is reduction, where all of the responses from the collection are combined using a summary function such as addition or maximum value.

We suggest supporting reduction patterns via an additional graphical primitive such as the one shown in figure 6. In this example suppliers is a collection of price databases that produce bids on orders from a customer. All of the bids in the collection are combined in a reduction that chooses the minimum value, resulting in the lowest bid. The interactions needed for price databases to enter and leave the collection are not shown; they have a structure similar to Fig. 5.

6 Conclusions

CAPS has been used to develop a variety of prototypes, including an architecture for a generic command and control station [6]. Our experience indicates that abstraction and generalization are essential for realizing the full benefits of systematic approaches to software architecture in the context of prototyping. The purpose of an architecture is to achieve system integration while preserving flexibility of system behavior. Human understanding plays an important role in the design of a software architecture, and simplicity of representation can help to use this scarce resource as effectively as possible.

Good software architectures should be able to accommodate most of the evolutionary changes to a software system and most of the alternative configurations in a system family with little or no impact on the architecture, by replacing selected components that fit into the architecture. This requires explicit design of the degrees of freedom to be supported by an architecture as well as the structures and constraints that enable an architecture to provide given types of capabilities and services. Thus the slots in an architecture should have general specifications that are compatible with many different components that can fit into the slot, while still ensuring that all components that fit will be able to work together in harmony to achieve the overall goal of the architecture.

Automated decision support is needed to make this work smoothly. Basic capabilities of a mature supporting environment include automated reuse and code generation capabilities for

both components and interconnections. More advanced capabilities include support for evolution of architectures, design team coordination, and analysis and testing support for both components and connection patterns to create effective architectures and to assess their effectiveness.

References

[1] Special issue on software architectures, *IEEE Trans. on Software Eng. 21*, 4, (April 1995).

[2] S. Badr and Luqi, Automation Support for Concurrent Software Engineering, *Proc. of the Sixth International Conference on Software Engineering and Knowledge Engineering*, Jurmala, Latvia, June 20-23, 1994, pp. 46–53.

[3] V. Berzins and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991, ISBN 0-201-08004-4.

[4] Luqi, Automated Prototyping and Data Translation, *Journal of Data and Knowledge Engineering 5*, (July 1990), pp. 167–177.

[5] Luqi, A Graph Model for Software Evolution, *IEEE Trans. on Software Eng. 16*, 8, (Aug. 1990), pp. 917–927.

[6] Luqi, Computer-Aided Prototyping for a Command-and-Control System Using CAPS, *IEEE Software 9*, 1, (Jan. 1992), pp. 56–67.

[7] Proc. of Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Evolution Monterey, CA, Sep. 1994.

[8] Special Issue on CAPS, *Journal of Systems Integration 6*, 1/2, (1995).

[9] W. Myers, Taligent's CommonPoint: The Promise of Objects, *Computer 28*, 3 (March 1995), pp. 78–83.

[10] D. Parnas, On the Criteria to be Used in Decomposing a System into Modules, *CACM 15*, 12 Dec. 1972, pp. 1053–1058.

[11] B. Ramesh and Luqi, Process Knowledge Based Rapid Prototyping for Requirements Engineering, *Proceedings of IEEE/ACM Symposium on Requirements Engineering*, San Diego, CA, Jan. 1993, pp. 248–255.

[12] R. Steigerwald, Luqi and J. McDowell, CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping, *Information and Software Technology 33*, 9, Nov. 1991, pp. 698–706.

[13] G. Wiederhold, P. Wegner, S. Ceri Toward Megaprogramming, *CAM 35*, 11, (Nov. 1992), pp. 89–99.

Parameterized Programming and Software Architecture*

Joseph A. Goguen
Programming Research Group, Oxford University Computing Lab

Abstract: This paper discusses an approach to software architecture based on concepts from parameterized programming, particularly its language of "module expressions." A module expression describes the architecture of a system as an interconnection of component modules, and executing the expression actually builds the system. Language features include: modules parameterized by theories, which declare interfaces; a number of module composition operations; views for binding modules to interfaces; and both vertical and horizontal composition. Modules may involve information hiding, theories may declare semantic restrictions with axioms, and views assert behavioral satisfaction of axioms by a module. Some "Laws of Software Composition" are given, showing how various module composition operations are related. The paper also shows how a variety of communication styles can be supported in this approach, and how it can be extended to provide support for evolution and traceability. All this is intended to ease the development of large systems, and in particular, to make reuse more effective in practice.

1 Introduction

This paper presents an approach to software architecture that is based on concepts from parameterized programming. Parameterized programming [5, 6] concerns design and module composition times, rather than compile or run times; this is, it addresses the architectural level of software. It supports building systems, software reuse, and controlled evolution, as well as the management of configurations, versions, families, documentation, etc. The "module expressions" used in parameterized programming constitute a module connection language (abbreviated MCL, and sometimes also called an architecture description language, or ADL). A module composition language may be used

1. *descriptively*, to specify and analyze given design, or
2. *constructively*, to describe a new design using existing modules, and execute it to build a new system.

Detailed design and coding are unnecessary for construction or description if a suitable database (hereafter called a library) of specifications and relationships among them is available. Parameterized programming supports both construction and description, assuming availability of a library that contains:

1. **module expressions**, describing systems as interconnections of modules, and
2. a **module graph**, describing modules and relationships among them.

*The research reported in this paper has been supported in part by the Office of Naval Research, the Ada Joint Project Agency, ARPA as part of its DSSA (Domain Specific Software Architecture) project, the UK Science and Engineering Research Council, the CEC under ESPRIT-2 BRA Working Groups 6071, IS-CORE (Information Systems COrrectness and REusability), and 6112, COMPASS (COMPrehensive Algebraic Approach to System Specification and development), Fujitsu Laboratories Limited, and the Information Technology Promotion Agency, Japan, as part of the R and D of Basic Technology for Future Industries "New Models for Software Architecture" project sponsored by NEDO (New Energy and Industrial Technology Development Organization).

A module graph can incorporate executable code for modules, as well as specifications, and other software objects. When a module expression is executed in the presence of a suitable module graph, an executable system can be constructed by manipulating and linking the implementation modules.

The parameterized programming approach to software architecture has been validated by experience with LILEANNA [21], an MCL using Ada for implementation and Anna [15, 16] for specification. LILEANNA was developed as part of the DSSA ADAGE project sponsored by ARPA. The implementation was done by Will Tracz of Loral Federal Systems, and has been used for helicoptor navigation software. This approach seems especially useful for "software factory" situations, such as the Loral helicopter navigation software facility, where a number of similar systems are produced over time. In such cases, systems like LILEANNA may be able to save a great deal of software development time, although significant initial investment may be needed to accumulate information for the library.

LILEANNA has a formal semantics based on category theory, following ideas developed for the Clear specification language [3, 4]; more recently, a set theoretic semantics has been given [12]. These semantics are very general, and apply to languages other than Ada and Anna, and indeed, to any implementation-specification language pair that satisfies certain axioms. The properties of an interconnection of modules are related in a straightforward way to those of its components, because of the precise and straightforward semantics of module expressions. The work reported in this paper is based on ideas developed in 1983, and first reported in [5], which suggested a design for LIL, a library interconnection language for Ada; see also [6].

1.1 What is Architecture?

The term "architecture" has been much discussed in recent literature, and there is now a large family of partially overlapping definitions. This situation suggests that instead of arguing over the meaning a single popular term, we should develop new terminology that distinguishes among the most important concepts of software architecture. This subsection suggests one such terminology, following ideas from parameterized programming.

A narrow meaning of architecture concerns *static* aspects of systems, including structure, components, relationships among components, communication style, etc. A wider sense concerns the *dynamic* aspects of software development, such as the (ever changing) rationales for design choices, and their traceability back to the (ever changing) requirements (the importance of requirements and their evolution is further discussed in [8]). We may call these **static** and **dynamic architectures**, respectively. Some aspects of dynamic architecture are discussed in Section 5.

It is also important to distinguish the architecture of a particular system from the knowledge needed for constructing a family of related systems (or developing a large complex evolving system). We suggest calling these **system architecture** and **domain architecture**[1], respectively. In parameterized programming, a module expression captures a particular design, while a module graph captures an architectural domain. Both are needed to support large scale system development efforts.

Acknowledgements

I thank Will Tracz for many useful discussions, and for help with the examples in this paper.

[1]The word **framework** is sometimes used for a collection of modules capturing common aspects of applications over a certain problem domain.

2 Hyperprogramming and Module Graphs

Parameterized programming[2] assumes that all modules have associated specifications; these serve as "headers" for other information, particularly source code and compiled code. The specifications need not be complete, but must include at least the syntax of what is exported by the module, the syntax of its interface if it is parameterized, and the names of any modules that it imports. In parameterized programming, specification and code modules are collected together with other information to form a module graph, which describes the organization of the system development database, including information relevant to both system and domain architecture[3].

A node of a module graph may have as header a

- theory specification,
- package specification, or
- module expression,

while an edges of the module graph may be labelled with a module relationship, such as

- inheritance,
- view,
- parameterization, or
- instantiation.

Package specifications are intended to have associated executable code. Theory specifications are not, as discussed in the next subsection. Other software objects that maybe be associated with nodes or edges are also discussed below.

The organization of module graphs is based on some ideas from what we call **hyperprogramming** [7]. The most relevant ideas can be summarized as follows:

- modules are associated with **clusters**, where
- a specifications serve as *headers* for each cluster, and
- one or more implementations may be given for each package specification, including
  - source code, and
  - compiled code,
- plus (optionally)
  - test cases,
  - performance data,
  - documentation,
  - administrative information, such as programmer, date of last change, etc.,
  - rationale, and
  - links to other software objects.

Each cluster has its own node in a module graph. The other software objects referred to in the last point above might be requirements, dataflow diagrams, transcripts of interviews, review documents, etc.; these will also be attached to clusters associated with nodes in the module graph.

Here is a LILEANNA package specification for a stack of integers:

[2]The term *megaprogramming* is used for some rather similar ideas within the ARPA community [2, 22].

[3]The module graph is an *abstraction* of this organization. For various reasons, including efficiency and the structure of existing database systems, the information structure that is actually implemented may be quite different from that suggested by the mathematical structure of a graph.

```
package INTSTACK is
   import INTEGER;
   type Stack;
   exception Stack_Empty;
   exception Stack_Overflow;
   function Is_Empty(S: Stack) return Boolean;
   function Is_Full(S: Stack) return Boolean;
   function Push(I: Integer; S: Stack) return Stack;
  --| where
  --|       Is_Full(S) => raise Stack_Overflow;
   function Pop(S: Stack) return Stack;
  --| where
  --|       Is_Empty(S) => raise Stack_Empty;
   function Top(S: Stack) return Integer;
  --| where
  --|       Is_Empty(S) => raise Stack_Empty;
  --| axiom
  --|       for all I: Integer, S: Stack =>
  --|           if not Is_Full(S) then
  --|               Pop(Push(I, S)) = S and ;
  --|               Top(Push(I, S)) = I;
  --|           end if;
end INTSTACK;
```

Specifications need only be developed to the extent that it is practically useful to do so; often just the syntax is given. The above specification is incomplete, in that it does not define the functions Is_Full or Is_Empty.

2.1 Theories and Views

Theory specifications, called **theories** for short, are used to describe generic module interfaces; these may contain axioms, which serve as semantic constraints on the actual modules that are allowed by the interface. There are no implementation modules associated with theories.

The simplest theory just says that a type should be provided:

```
theory TRIV is
   type Element;
end TRIV;
```

Any (non-empty) module can be matched with TRIV, by designating one of its types.

The next theory has some axioms that provide semantic constraints on what can fit the interface that it defines, saying that the type should have a partial order structure:

```
theory POSET is
   type Element;
   function <= (X,Y: Element) return Element;
  --| axiom
  --|   for all X,Y,Z: Element =>
  --|     X <= X and;
  --|     if X <= Y then Y <= X end if; and;
  --|     if X <= Y and Y <= Z then
  --|       X <= Z end if;
end POSET;
```

Views are used to bind actual modules to generic module interfaces, in order to instantiate generics. Because modules may involve information hiding, views only assert the *behavioral* sat-

isfaction of the axioms in their source theory by the target module[4]. However, *proving* that such axioms are satisfied should not be part of a system intended to be practical for ordinary use. Instead, views are used for recording a programmer's belief that the axioms in the source theory hold in the target module. Support for such a belief may be provided off line on the back of an envelope, by giving a formal mechanical proof, or by anything between these extremes; the belief may even be left unsupported, although this is not recommended. The nature of the support given (e.g., a scanned image of the envelope back, or the source file of the machine proof) can be stored with the view in the module graph.

The view below asserts that the relation => on integers satisfies the axioms of POSET; it can be used to instantiate the generic LILEANNA package SORT given later, yielding a package with a function that sorts integers in *descending* order.

```
view GEQ :: POSET => Standard is
   types (Element => Integer);
   ops ("<=" => ">=");
end GEQ;
```

Another use of views is to assert global properties of systems, by giving a view to a top level module (represented by a module expression saying how the system is composed from lower level modules) from a theory with axioms giving the properties to be asserted. As before, these axioms need only be behaviorally satisfied.

3 Parameterized Modules

The most characteristic feature of parameterized programming is the parameterization of modules over interface declarations: any module that "fits" the interface can be "substituted into" the parameterized module, yielding a new module that is an "instance" (or "instantiation") of the original module. The interface is described by a theory.

Below is LILEANNA code for a parameterized version of the INTSTACK module given earlier (material that is the same as in INTSTACK is indicated by .....):

```
generic package STACK[Element :: TRIV] is
   type Stack;
   .....
   function Push(E: Element; S: Stack) return Stack;
   .....
   function Top(S: Stack) return Element;
   .....
   -- | axiom
   -- |   for all E: Element, S: Stack =>
   -- | .....
end STACK;
```

The generic package specification below is for a sorting program, parameterized by the POSET theory, which says that a partially ordered set should be supplied in order for the program to work correctly:

```
generic package SORT[Item :: POSET] is
   importing LIST[Item];
   function Sort (X: List) return List;
   function Sorted (X,Y: List) return Bool;
```

[4]This means that the axioms need only appear to hold under all possible "experiments" on the module, involving its externally visible operations; sometimes this is also called *observational satisfaction.*

```
  -- | axiom
  -- |   for all X: List =>
  -- |     Sorted(Sort(X)) = true and;
  -- |     ..... ;
end SORT;
```

3.1 Instantiation

The SORT program specified above can be instantiated with the view GEQ simply by writing SORT[GEQ]; this results in a program for sorting integers in descending order. Similarly, we could instantiate SORT with the partial ordering relation of divisibility on natural numbers by writing SORT[DIV], where DIV is a suitable view.

Default views enable "obvious" views to be replaced by the name of their target module, or even the name of a type. For example, we would not have to write out a view from POSET to Standard that mapped the type Elt to Integer and the operation symbol => to itself, but could just write SORT[Integer]. Default views are computed using a certain set of default rules that are given in [6], and that capture many of our intuitions about what is "obvious". The following are some instantiations that use default views:

```
STACK[Integer]
STACK[LIST[Integer]]
STACK[STACK[Float]] .
```

(Default views were first implemented in OBJ3 [13], and are partially implemented in LILEANNA.)

An interface theory can call for more than one operation, and more than one type, and views can be written that express bindings to such interfaces. Generic modules can also have more than one interface, each defined by its own theory. These will require more than one view for instantiation.

3.2 Vertical Composition

Vertical structure describes the use of lower layers (virtual machines), whereas *horizontal* structure describes a given layer; the distinction between vertical and horizontal structure was first named and formalised by Goguen and Burstall [9]. Parameterized programming provides parameterization and instantiation for both vertical and horizontal structure. The following generic package specification has one horizontal and one vertical parameter:

```
generic package SORT[Item :: POSET](LISTP :: LIST[Item]) is
  function Sort (X: List) return List;
  function Sorted (X,Y: List) return Bool;
  -- | axiom
  -- |   for all X: List =>
  -- |     Sorted(Sort(X)) = true and;
  -- |     ..... ;
end SORT;
```

Note that the horizontal interface theory is itself parameterized, and moreover is instantiated with the horizontal formal parameter. There is also a default view from TRIV to POSET involved in the vertical instantiation, since SORT wants a POSET whereas LIST wants a TRIV. Also note that the notation [_] is used for horizontal parameterization, while (_) is used for veritcal parameterization. The same conventions apply to instantiation, as illustrated by the following module expression (where each instantiation uses a default view):

```
SORT[Natural](LIST7[Natural]) .
```

Modules can also import (or "inherit") other modules. The simple syntax for this is illustrated in the following:

```
package M42 is
    inherits SYS31;
    inherits Sort[Integer];
      ..... ;
end M42;
```

Inherited submodules are always shared, that is, new copies are not produced.

Figure 1 is a graphical evocation of the relationships among inheritance, horizontal parameterization, and vertical parameterization. Note that under horizontal instantiation, actual parameters are shared, whereas under vertical, new copies are used. Here M is a module, I is an imported module, T_H is a horizontal interface theory, and T_V is a vertical interface theory.

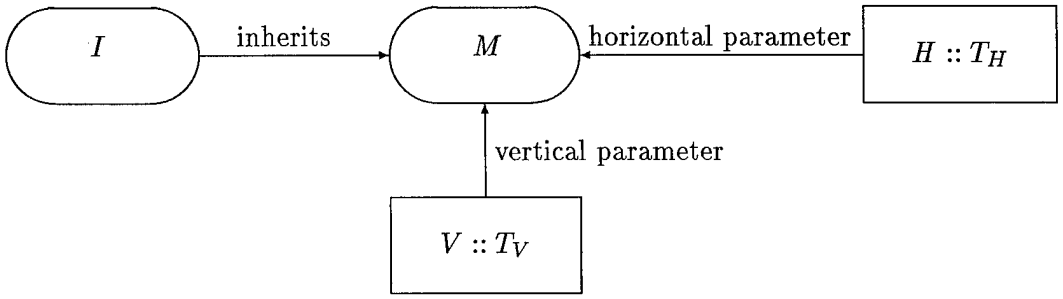


Figure 1: Horizontal and Vertical Composition

4 Module Expressions

The purpose of module expressions is to define software architectures. Therefore module expressions need more than parameterization and instantiation. The following additional operations for combining modules are implemented in LILEANNA:

- module aggregation, i.e., simple combination, taking account of
  - shared inherited modules, and
  - parameterization;
- deleting functionality;
- modifying functionality; and
- adding functionality.

Both operations and types can be renamed, deleted or added, as can exceptions, axioms, etc. And of course, both horizontal and vertical instantiation are allowed. The following module expressions illustrate the syntax:

```
SYS1 + SYS2
SYS1 * (rename op Put => Write) + SYS2 * (delete op Undo)
STACK * (rename type Stack => Stack1)[Integer]
  + STACK * (rename type Stack => Stack1)[LIST[Float]]
```

Thus + "adds" modules, i.e., forms a system containing all the summands; because summation is associative (see Section 4.2), any number of summands can be given without using parantheses. See [1] for some related work.

The **make** command "executes" a module expression to actually build a (sub)system, give it a name, and store it in the environment. The following illustrate this:

make SYS3 **is** STACK[STACK[Float] + SYS2 * (**delete op** Undo) **end** SYS3;

A **make** can also be parameterized, as in the following:

make SYS4[X :: POSET] **is** SORT[X](LIST12[X]) **end** SYS4;

Of course, a make command can use results of previous makes, as in

make SYS5 **is** SYS3 + SYS4[Float] **end** SYS5;

Furthermore, a **make** statement can be used to impose sharing of parameters on its constituents, as in the following:

make M329[X :: POSET] **is** SORT[X](LIST12[X]) + STACK[X] **end** M329;

Whenever a module expression is introduced, it is added as a node to the module graph, and whenever a module expression is evaluated, executable code is generated and attached to the cluster of the module expression.

Module expression evaluation can be implemented by manipulating intermediate compiled code (for LILEANNA, this is Ada's intermediate compiled code language DIANA). Intermediate compiled code is easier to manipulate than object code, while source code does not have much of the required information in a sufficiently explicit form. One also gets the benefit of being able to put the manipulated code through the compiler's backend, including optimization. Module expressions were first implemented in the OBJ system [13], but using different techniques, because there is no attached code in a separate programming language. The mathematical semantics of module expression evaluation is given by the colimit of a diagram extracted from the module graph [4].

LILEANNA provides a graphical "layout editor" for module expressions, based on notation like that typically used by engineers, i.e., boxes and arrows. This helps users to edit existing module expressions to define new (sub)systems, or define them from scratch, as they wish.

4.1 Architectural Styles

Many different architectural styles can be supported by parameterized programming, including different communication styles, such as shared variables, pipelining, message passing, and blackboarding. These can be described by using shared submodules in various ways. (Such a cell is a simple "object" in the sense of object oriented programming, and can be easily specified in LILEANNA.) For example, if a "cell" module C encapsulates a variable X, and if modules F and G each inherit C, then they share X, and in the system F + G, they can communicate through it. Similarly, a "post office" module can be inherited by a set of modules, and then used for passing messages among them.

Pipelining is a special case of shared variable communication, where only two modules share each cell; the cells represent the pipes, with one of the importing module reading the variable, and the other one writing it. For example, if modules F and G import a cell C1 and modules G and H import a cell C2, then F + G + H can function as a (short) pipeline.

The following subsection sketches ways to describe avionics architectures using module expressions. Here the modules encapsulate various digital signal processing subsystems, and shared cells represent "wires" that pass the signals.

4.1.1 Describing Avionics Systems with Parameterised Programming

An interesting example in this domain of a module that takes other modules as parameters is a Kalman filter module K that needs a model A of the aircraft; this situation is expressed by the simple module expression K[A]. It makes sense to parameterize K by aircraft models, so that the same filter can be used for many different models.

Now consider a flight control system F that needs a guidance system G, where G needs an aircomputer A. We can describe this situation with the module expression F[G[A]]. (We omit details of the code here and hereafter.) Assuming that the modules F and G have parameter (interface) theories GS and AC respectively, then views are needed to match A to AC and G to GS; it is natural to expect that default views will work in such examples.

This simple approach based on instantiating parameterized modules works well if the system architecture is linear, or more generally, a tree; but it is not adequate for sharing among more than two modules, or for feedback loops. Feedback is necessary in avionics software, because feedback control is a crucial technique.

For example, suppose the value of a floating point variable X in A needs to be fedback into F. We can capture this by encapsulating X in C, and letting F and A each inherit C; this system is still described by the simple module expression F[G[A]]. Another approach is to provide each F and A with a new parameter for a floating point cell, in which case the system is described by the module expression F[G[A[C]], C].

This example highlights the importance of sharing for a module interconnect formalism. We have shown that parameterised programming can accomplish such sharing in several ways. Without such a capability, it would be difficult or impossible to handle feedback and variables shared among several modules.

4.2 Some Software Laws

There are many relationships among the various operations on modules; the more important of these can be considered "laws of software." Here are a few of them:

$$M + M' = M' + M \ ,$$

$$M + (M' + M'') = (M + M') + M'' \ ,$$

$$M + M = M \ ,$$

$$M + N = M \text{ if } N \text{ inherits } M \ ,$$

$$M^H + M'^H = (M + M')^H \ ,$$

$$M^{N_1,N_2} = M^{N_2,N_1} = M^{N_1+N_2} \ ,$$

$$M^{N_1^H,\ N_2^H} = M^{(N_1+N_2)^H} \ ,$$

$$M^N + N = M^N \ ,$$

where M^N indicates that M inherits the module N, and M^H indicates that M inherits a set H of modules. There are many more laws, e.g., for parameterization; see [12] for these, as well as for proofs. These laws are used in the implementation of LILEANNA for simplifying module expressions.

5 Support for Evolution and Traceability

A traditional view is that software evolution only occurs after initial development is complete. By contrast, we consider evolution to include all activities that change a system, as well as the relationships among those activities, occurring throughout the system's life. Thus, the term "evolution" focuses attention on *change*, which is inevitable and unending throughout software development. Moreover, since large complex systems are inevitably embedded in complex evolving social contexts, they will necessarily *co-evolve* with those contexts, in the sense that each will affect the evolution of the other. (See [10] for further discussion of change and social context.)

The ubiquity of change motivates the use of iterative lifecycle processes, and especially prototyping, i.e., quickly building and evaluating a series of prototypes, which are concrete executable models of selected aspects of a system [17]. The ability of parameterized programming to describe software architectures, in both the domain and systems senses, can greatly facilitate prototyping. In some cases, all that need be done is edit a module expression. In other cases, the module graph may need updating, e.g., writing new modules or modifying old ones. Then executing the module expression yields a running prototype, which can be gracefully evolved into the actual system, and thereafter further evolved.

The additional information needed to cope with the dynamic evolution of (families of) software systems is provided by enriching the module graph with relevant relationships among various software objects, such as requirements and rationales. Rationales should be considered part of evolution support rather than architecture, because they must evolve along with the objects that they concern.

5.1 Traceability

The Centre for Requirements and Foundations at Oxford has projects to improve the acquisition, traceability, accessibility, modularity, and reusability of the numerous objects that arise and are manipulated during software development, with a particular focus on the role of requirements. An initial study [14] administered a detailed two-stage questionnaire to software engineers at a large firm, and found that traceability was considered the most important outstanding problem. Further analysis showed that there are actually several different traceability problems, which should be treated in different ways. Major distinctions are between pre-RS (Requirements Specification) traceability and post-RS traceability, and between forward and backward traceability.

Tracing back as far as requirements is important for developing large software systems. But it is also difficult because of the overhead of maintaining the huge mass of dependencies among the many objects produced by a large software development effort. Moreover, dependencies reaching far across the development cycle can be significant. Without formal representations for the objects involved, formal models for the dependencies, and tool support for managing them, it can be impossible to know what effect a change will have, and in particular, to know what other objects may have to be changed to maintain consistency. A hypergraph model for maintaining evolving dependencies, suitable as a basis for tool development, is given in [18]; this structure can be applied to give a model for evolving module graphs. Work on capturing domain knowledge has used methods from sociology, particularly ethnomethodology and its disciplines of conversastion and interaction analyses. See [8, 10] for further information on this research.

We are also developing a system called TOOR to support tracing dependencies among evolving objects, and in particular, to show how decisions are grounded in prior objects [19]. Significant subproblems include formalizing dependencies, and developing methods for calculating dependencies and for propagating the implications of changes. This approach, called *hyperrequirements*, builds on parameterized programming and hyperprogrammming, and is intended to support the social context of decisions, as well as their traceability, by linking related objects, based on the broad view of context and requirements suggested in [8].

TOOR supports user-definable relations, to allow differentiating among different links between the same objects. It also allows declaring mathematical properties of relations, such as transitivity, to give additional power and flexibility in tracing links through specifiable compositions of relations. TOOR supports several different trace modes, including browsing and regular expression based search, and it also supports module definition through an intuitive template-driven graphical interface. Hyperprogramming and hyperrequirements support reuse, through the generalized

notion of relation for linking objects for requirements, design, specification, coding, documentation, maintenance, etc.

TOOR is built on FOOPS [11, 20], a general object oriented language with specification capabilities. This makes TOOR particularly suitable for use in an object oriented development paradigm. TOOR uses FOOPS-like modules to declare software objects and relations, and to automatically create links as objects are interconnected and evolve. TOOR also provides hypermedia facilities, based on HTML, to make its use closer to analysts' intuitions and natural activities. For example, graphs, charts, and videos can be linked, as well as conventional documents.

6 Discussion and Conclusions

We have seen that parameterized programming can provide a systematic approach to software architecture, through its use of theories and views, its powerful methods for combining and modifying components to form new systems, and its underlying module graph data structure. The notions of module expression and module graph, we believe, add some clarity to discussions about the nature of software architecture. We have illustrated the use of module expressions to achieve several different architectural styles.

Parameterized programming extends to hyperprogramming and hyperrequirements, to support evolution by including other software objects in the module graph, such as requirements, rationales, and documentation, as well as relations to support traceability. This enables design objects and relations to be managed in a systematic way, which can have a significant impact on the reusability of code, through the reuse of design information. Support for evolution also means that prototyping and traceability integrate with the approach in a natural way. Development system integration and enhanced reusability are the result of a systematic way of storing information in an evolving module graph. It is hoped that ideas like these will make the development of large complex systems more reliable and efficient.

References

[1] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. Technical Report TR–91–22, Department of Computer Sciences, University of Texas, Austin, 1991. Revised May 1992.

[2] Barry W. Boehm and William L. Scherlis. Megaprogramming. In *Proceedings of Software Technology Conference 1992*, pages 63–82, April 1992.

[3] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[4] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland, 1978.

[5] Joseph Goguen. Suggestions for using and organizing libraries in software development. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, First International Confer-*

ence on Supercomputing Systems, pages 349–360. IEEE Computer Society, 1985. Also in *Supercomputing Systems*, Steven and Svetlana Kartashev, Eds., Elsevier, 1986.

[6] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[7] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology*. Joint System Development Corporation, Tokyo, Japan, January 1990.

[8] Joseph Goguen. Requirements engineering as the reconciliation of social and technical issues. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–200. Academic Press, 1994.

[9] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report Report CSL-118, SRI Computer Science Lab, October 1980.

[10] Joseph Goguen and Luqi. Formal methods and social context in software development. In Peter Mosses, Mogens Nielsen, and Michael Schwartzbach, editors, *Proceedings, Sixth International Joint Conference on Theory and Practice of Software Development (TAPSOFT 95)*, pages 62–81. Springer, 1995. Lecture Notes in Computer Science, Volume 915.

[11] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153–162, October 1986.

[12] Joseph Goguen and Will Tracz. An implementation-oriented semantics for module composition, 1995. In preparation.

[13] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies*. Cambridge, to appear. Also Technical Report, SRI International.

[14] Orlena Gotel. Requirements traceability. Technical report, Centre for Requirements and Foundations, Oxford University Computing Lab, December 1992.

[15] Bernd Krieg-Brückner and David Luckham. ANNA: Towards a language for annotating Ada programs. *SIGPLAN Notices*, 15(11):128–138, November 1980.

[16] David Luckham. *Programming with Specifications: An Introduction to* ANNA, *A Language for Annotating Ada Programs*. Springer, 1990.

[17] Luqi. Software evolution through rapid prototyping. *IEEE Computer*, 22(5):13–25, 1989.

[18] Luqi and Joseph Goguen. Formal methods: Problems and promises. *IEEE Software*, 1996. To appear.

[19] Francisco Pinheiro and Joseph Goguen. Design and use of an object-oriented tool for tracing requirements. *IEEE Software*, to appear, March 1996. Special issue of papers from ICRE '96.

[20] Adolfo Socorro. *Design, Implementation, and Evaluation of a Declarative Object Oriented Language.* PhD thesis, Programming Research Group, Oxford University, 1994.

[21] Will Tracz. Parameterized programming in LILEANNA. In *Proceedings, Second International Workshop on Software Reuse*, March 1993. Lucca, Italy.

[22] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward megaprogramming. *Communications of the ACM*, 35(11):89–99, 1992.

Specification Merging for Software Architectures

Major David A. Dampier, U. S. Army
Lieutenant Colonel Ronald B. Byrnes, U. S. Army
U.S. Army Research Laboratory, Software Technology Branch
115 O'Keefe Building
Atlanta, GA, 30332-0862, USA
dampier@airmics.gatech.edu
byrnes@airmics.gatech.edu

Abstract

Building large, complex software systems can no longer be accomplished through brute force methods with an unlimited number of programmers. Budget constraints both on time and people require the development of more sophisticated formal methods for software development and evolution.[8] Two ways to increase the efficiency of software development and maintenance are rapid prototyping and reuse. Rapid prototyping is used to validate requirements for the system through user feedback, and reusable components provide the common functionality for the software. These reusable components would be formally specified and stored in a reuse repository for easy retrieval. To make this solution tractable, reusable components must be organized into groups of components for specific purposes. A common name for this idea is Domain-Specific Software Architectures (DSSA). This paper discusses merging, one method of aiding in the evolution of software prototypes. Merging is used to combine the effects of different changes to software components during evolution, and to ensure consistency between independent enhancements during evolution. Automated methods are becoming available to merge many different types of programs. One such method, change-merging for data flow programs, is explained here along with a useful and necessary extension

1 Introduction

Building large and complex software systems is the focus of much new research, including a brand new ARPA program [11]. Automated methods for the evolution of large and complex systems is the focus of a renewed effort in the U.S. software development industry and in academia. These methods include rapid prototyping to help define the customer's requirements more precisely early in the evolutionary process, reengineering to salvage some of the vast amounts of legacy code currently in use, and reuse to make use of code that can be shared among different applications. The first and last of these methods set the stage for the focus of this paper.

Rapid prototyping is a method in which functional prototypes of a software system are built quickly and iteratively to allow customer feedback to drive the design. This feedback is used in each iteration of the process to redesign the prototype for additional review and feedback by the customer. This iterative review and redesign process continues until the customer is satisfied with the "look and feel" of the prototype. The result of this process is then a validated set of require-

ments and a baseline architecture of the system that can be used to develop the operational software.

2 Prototyping Software Architectures

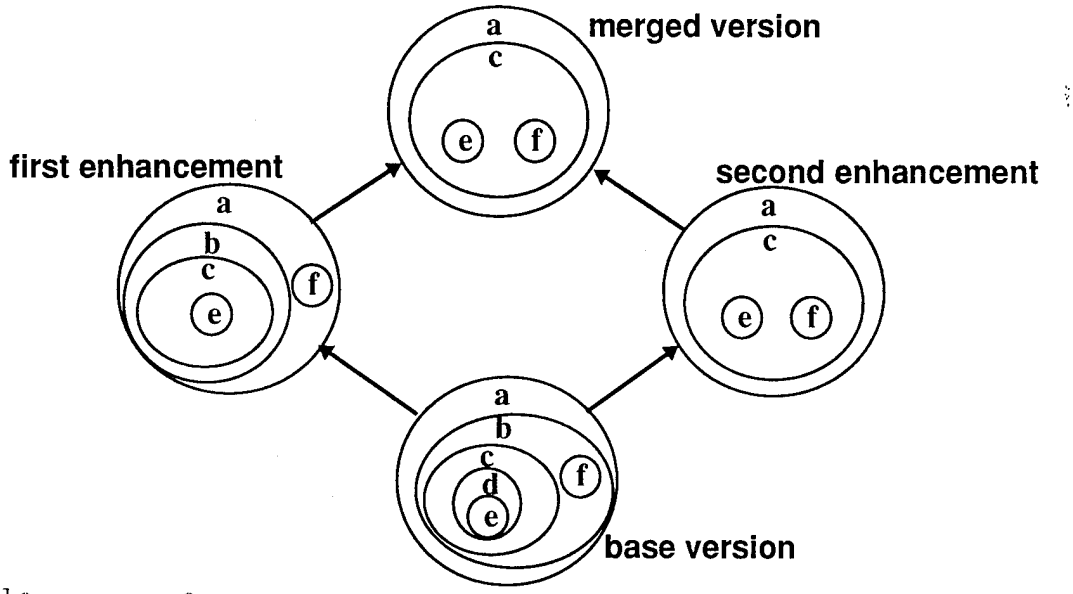
Evolutionary software prototypes like those developed using the CAPS rapid prototyping environment [9] are representations for architectures of the intended software system. They contain all of the necessary components of a software architecture as defined by this workshop: components, connections, and rationale. The components are defined by the operators and types in the prototype. The connections are defined by the data streams and timer interactions in the prototype. The rationale is defined by the control constraints. [10]

3 Merging Prototypes

Change-merging of software prototypes was thoroughly defined by Dampier in [5,6], and highlighted in earlier versions of this workshop [4,7]. Semantics-based change-merging [1,2] is essentially combining different changes to a software prototype by comparing the behavior of the different versions and merging the relevant behavioral changes, rather than structural changes. This is accomplished through prototype slicing [5] by comparing the slices of the modified versions of the prototype with the same slices of the base version. Where the slices differ, significant behavior changes are observed. These behavior changes in each of the modifications are combined with the preserved behavior from the base in both modifications to create the merged version. Slices of the merged version are then compared to slices of the modifications with respect to the affected part of each to ensure the significant behavior is preserved in the merged version. When these slices differ, a conflict is indicated. It can then be located by observing where the slices differ.

This change-merging method uses flattened implementation graphs of the software prototypes. This is necessary to properly capture all of the interactions in the prototype. Unfortunately, this method does not offer a way to reconstruct a hierarchically decomposed solution. In [3], Berzins and Dampier offer an extension to the original method that provides for change-merging the hierarchical decompositions as well. This extension stores the original decompositions for each input version as a set of ancestor chains. An ancestor chain for each component is the sequence of identifiers that represent the component's parent, then the parent's parent, and so on until the top-level prototype is reached. These ancestor chains can be merged using *greatest lower*

bound, least upper bound, and *pseudo-difference* operations from a Brouwerian algebra structure constructed from the partial ordering over all possible ancestor chains. Berzins and Dampier provide a formal model and extended algorithm for the change-merging method [3]. An example of a change-merge operation performed on hierarchical decompositions using the merging equation $A[B]C = (A \dot{-} B) \sqcup (A \sqcap C) \sqcup (C \dot{-} B)$ is contained in Figure 1.



a: <>[<>]<> = <>
b: <a>[<a>]<> = <>
c: <ab>[<ab>]<a> = <a>
e: <abc>[<abcd>]<ac>=(<abc> $\dot{-}$ <abcd>) $\sqcup$ (<abc> | |<ac>) $\sqcup$ (<ac> $\dot{-}$ <abcd>)= $\perp$ $\sqcup$ <a> $\sqcup$ <ac>=<ac>
f: <a>[<ab>]<ac> = ((<a> $\dot{-}$ <ab>) $\sqcup$ (<a> | |<ac>) $\sqcup$ (<ac> $\dot{-}$ <ab>)= $\perp$ $\sqcup$ <a> $\sqcup$ <ac>=<ac>

Figure 1. Example of a change-merge of decompositions [3].

The $\dot{-}$ represents the *pseudo-difference* operation, the $\sqcup$ represents the *least upper bound* operation, and $\sqcap$ represents the *greatest lower bound* operation. More detail on this extended change-merge method is available in [3].

4 Conclusion

This position paper explores the view that software prototypes are architectural representations for the intended software system and the possible change-merge operations that can be applied to those representations. These change-merge operations allow independent exploration of different architectural designs during the evolutionary prototyping process, with the assurance that semantic correctness can be guaranteed.

References

1. Berzins, V. "Software Merge: Semantics of Combining Changes to Programs", Submitted for publication in ACM Transactions on Programming Languages and Systems, 1990.

2. Berzins, V. "Software Merge: Models and Methods for Combining Changes to Programs", *Journal of Systems Integration*, vol. 1, no. 2, pp. 121-141, August 1991.

3. Berzins, V. and Dampier, D., "Software Merge: Combining Changes to Decompositions", to appear in *Journal of Systems Integration*, 1995.

4. Dampier, D. and Berzins, V., "A Slicing Method for Semantic Based Merging of Software Prototypes", *Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Slicing, Merging and Integration*, U.S. Naval Postgraduate School, Monterey, CA, October 1993, pp. 22-24.

5. Dampier, D., Luqi, Berzins, V., "Automated Merging of Software Prototypes", *Journal Of Systems Integration*, Kluwer Academic Publishers, March 1994.

6. Dampier, D., *A Formal Method for Semantics-Based Change-Merging of Software Prototypes*, Ph.D. Dissertation, U.S. Naval Postgraduate School, Monterey, CA, June 1994.

7. Dampier, D. and Berzins, V., "Software Change-Merging in Dynamic Evolution", *Proceedings of the 1994 Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Evolution Control for Large Software Systems,* U.S. Naval Postgraduate School, Monterey, CA, September 1994, pp. 17-20.

8. Feather, M., "Evolution of Specifications", *Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Slicing, Merging and Integration*, U.S. Naval Postgraduate School, Monterey, CA, October 1993, pp.38-40.

9. Luqi and Ketabchi, M., "A Computer Aided Prototyping System", *IEEE Software*, March 1988, pp. 66-72.

10. Luqi and Berzins, V., "Software Architectures in Computer-Aided Prototyping", *Proceedings of the 1995 Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Building Software with Specification Based Components*, U.S. Naval Postgraduate School, Monterey, CA, September 1995.

11. Waugh, D., "Evolutionary Design of Complex Software", *Proceedings of the 1995 Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Building Software with Specification Based Components, U.S. Naval Postgraduate School, Monterey, CA, September 1995.*

Formal Methods in Describing Architectures

Dr. Paul C. Clements
Software Engineering Institute[1]
Carnegie Mellon University
Pittsburgh, PA 15213

Introduction

Formal methods are gaining prominence in software engineering as a way to insure that a specification is consistent with its intended meaning, and that two formally-rendered artifacts (e.g., a specification and an implementation) are consistent with each other in some precise way. Formal methods in the arena of software architecture tend to manifest themselves in representation technology, principally in *architecture description languages* (ADLs). Rapide, UniCon, Wright, ACME, ArTek, RESOLVE, Gestalt, and other ADLs are populating the software architecture literature, each offering a formal way to represent the architecture of a software system.

But to what end? Formal methods are useful to help a human organize thought patterns into a more disciplined form, thus heading off conceptual errors. However, formal methods are most valuable when they precipitate automated checking of an artifact, or automated translation of an artifact from one form to a more useful form. Where do ADLs stand on these capabilities?

Modechart: A language on the edge of ADLs

This paper will present a language that stands at the edge of the world of ADLs. Modechart is a specification language for hard-real-time embedded computer systems developed at the University of Texas at Austin. For a complete overview, see [5]. Modechart was not invented to be an ADL; however, by imposing restrictions on its usage, it more than adequately represents software structures and the interactions among them. Why bother? Because Modechart features a sophisticated analysis environment that provides the user with a wealth of information about the system being specified, and the technology is based squarely on classical formal methods such as model-checking verification. Given a specification of the system, Modechart allows powerful analysis about the system itself (as opposed to checking the specification itself). We present the Modechart paradigm as an example of a fruitful trend for ADL research.

Introduction to Modechart

The Modechart user provides the following information in a specification:

- Modes: The user provides a set of modes and the contained in relation for those modes. A mode is either primitive, meaning it contains no other modes, serial, meaning that its immediate children may only be active one at a time in any nonzero time interval, or parallel, meaning that all of its immediate children are active when it is. The contained-in relation is transitive, antisymmetric, and antireflexive; it is modeled graphically by drawing boxes anywhere as long as no edge of one box intersects an edge of another.

1. This work is sponsored by the U. S. Department of Defense.

A mode represents a program state; its children represent logical subsets of the state. A mode is active if the state it represents is true. For instance, a mode might represent all states in which an aircraft's landing gear is down; it might have two children that would each be active according to whether or not the aircraft was on the ground.

However, the mode structure can clearly be used to represent software components. In this case, a mode being active corresponds to the software component being active and making progress. Parallel modes and serial modes have the obvious meaning with respect to software components.

- Actions: The user defines a set of actions. An action definition consists of a host mode assignment, a program that computes a value, a state variable (defined elsewhere) to which the computed value is assigned whenever the action completes, and a designation of the action as periodic or sporadic. A sporadic action is executed exactly once when its host mode is entered. A periodic action is executed when its host mode is entered, and then again each time the period interval passes. The user supplies the relevant timing information.

  Actions are the way to specify behavior, both in terms of what behavior is precipitated (by stating the predicate that the action changes) and when the behavior is precipitated (by associated it with a mode whose activation triggers the action). In software architecture terms, actions define the behavior of the modes, which for our purposes are now models of software components.

- Transitions: The user provides a set of mode transition expressions; each mode transition expression is associated with an ordered pair of modes. The mode transition defines a condition that causes the first mode of the pair to be exited and the second mode of the pair to be entered. The pair may consist of the same mode twice. The mode pairs are subject to syntactic constraints of Modechart; e.g., a transition between two sibling modes in parallel with each other is forbidden. Mode transitions are defined to take zero time in Modechart. A transition takes place either as a result of timing (the exited mode has been active for a specified length of time) or as the result of events (other modes becoming active or inactive, predicates changing value, external events occurring, or actions starting or completing).

  In architectural terms, transitions are the representation of the interconnection mechanisms between components (modes). Invocation is the mechanism most easily imagined, but others are possible.

- External events: The set of external events relevant to the system. For each one, a separation parameter s that promises the minimum separation time between two consecutive occurrences of the event. External events are those over which the system being modelled has no control; their occurrence appears completely nondeterministic, subject to the minimum separation constraint.

- State variables: An action assigns a (computed) value to a state variable. The user provides the set of state variables and their correspondence to an action. The user may provide an initial value for each variable.

The state space of the system at any instant of time consists of, the set of active modes, the values of all of the state variables, the external events that occur at that time, the status of each of the actions (started, in progress, completed, aborted) at that time; and the value of the system clock.

Time is modelled by the nonnegative integers in Modechart. Timing constraints are applied to actions' execution times, mode transition times, and the minimum separation between consecutive occurrences of each external event.

The Modechart environment allows other information to be provided by the user, but these things are all that the semantic model requires. Modechart is primarily a graphical language, although a textual form is also available; Figure 1 illustrates a trivial example.

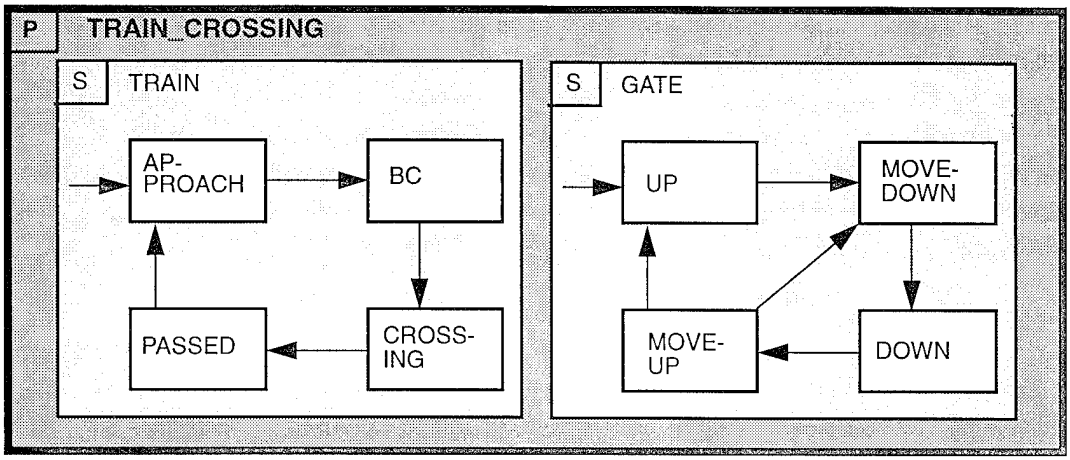


FIGURE 1. Graphical representation of a modechart. Non-atomic modes are tagged as being either serial (S) or parallel (P). Arrows may be labeled with the transition expressions that cause the transitions, but these are omitted here for readability.

The Modechart development and analysis environment

A suite of software engineering environment tools to facilitate use of the Modechart language is under development at the University of Texas at Austin [7], as well as at the U. S. Naval Research Laboratory in Washington [1] [2] [3]. A software engineer will have at his or her disposal the following major facilities available under the Modechart environment:

- a sophisticated graphical user interface for entering a modechart, displaying it, searching it, navigating through it, modifying it, and analyzing it [4];
- a simulator for observing an execution of a system modelled by the Modechart specification;
- a verifier for checking the consistency of the modechart against a logical assertion that is desired to be invariant [6] [8];
- auxiliary tools for converting modecharts to and from textual form, a database for storing modecharts, and a suite of consistency and completeness checkers to provide user feedback about the properties of the specification; and
- tools for entering a real-time systems specification in a language other than Modechart and having it translated automatically into a semantically consistent modechart.

Future work will encompass synthesis, the automatic generation of executable programs from

a Modechart specification.

The Modechart simulator

To run the simulator, the user first assembles a file of options (or uses a default set) that tells the simulation tool how nondeterministic decisions are to be made. These decisions include choosing a transition to take when more than one is enabled, scheduling the next time a randomly-occurring external event is to occur, and fixing the execution time of an action between given bounds. For example, although external events may occur any time after a specified delay (also referred to as a minimum separation), the user may instruct the simulator to model these events at specific times, at random times with given statistical distributions, or never. The user also specifies how long he or she wishes the simulation to run, the set of objects (e.g., modes) to be displayed, and a set of breakpoints. The simulator uses breakpoints much like a symbolic debugger. When a breakpoint occurs, the simulator halts to allow the user to inspect or to alter the computation. Breakpoints may occur at specified times or time multiples or they may coincide with events, e.g., whenever a particular mode is entered.

The simulator produces a bar graph. Each bar represents the behavior of a single Modechart object over time, where time begins at zero and is displayed horizontally from left to right. For modes, a thick line indicates that the mode is active at a particular time; a thin line indicates that it is not active. The simulator can also display the temporal behavior of external events, actions, transitions, and boolean variables.

The simulator is a powerful tool during specification creation and refinement, because it lets the user think in terms of behaviors of the system under design. However, it has the drawback that it can only exhibit one behavior of the system at a time, although the user has a great deal of influence about the choice of behavior illustrated.

The Modechart verifier

A more powerful tool for analysis is the Modechart verifier. The verifier allows a user to pose an assertion in a first-order predicate logic (enhanced in a minor way with the addition of an event occurrence function). The verifier then compares that assertion with a computation graph for the specification. A computation graph is a tree of all possible legal states of the system. The verifier checks to see if the assertion is true in all states, in some states, or in no states.

The verifier comes with a set of "pre-packaged" assertions that the user can inquire about. These assertions are suitably parameterized; they include:

- asking the maximum or minimum duration of a particular mode;
- asking whether two modes are ever active simultaneously;
- asking whether an activation of a particular mode is always surrounded in time by the activation of another particular mode; or
- asking whether a particular state is reachable (i.e., ever occurs in any computation);

The last query is particularly powerful. By phrasing an undesirable property in terms of a Modechart state and then inquiring as to that state's reachability, a user can quickly perform a safety analysis on the specification. If the undesirable state is not reachable, then the user is

guaranteed that no computation of a system that correctly implements the architecture will produce the undesirable state.

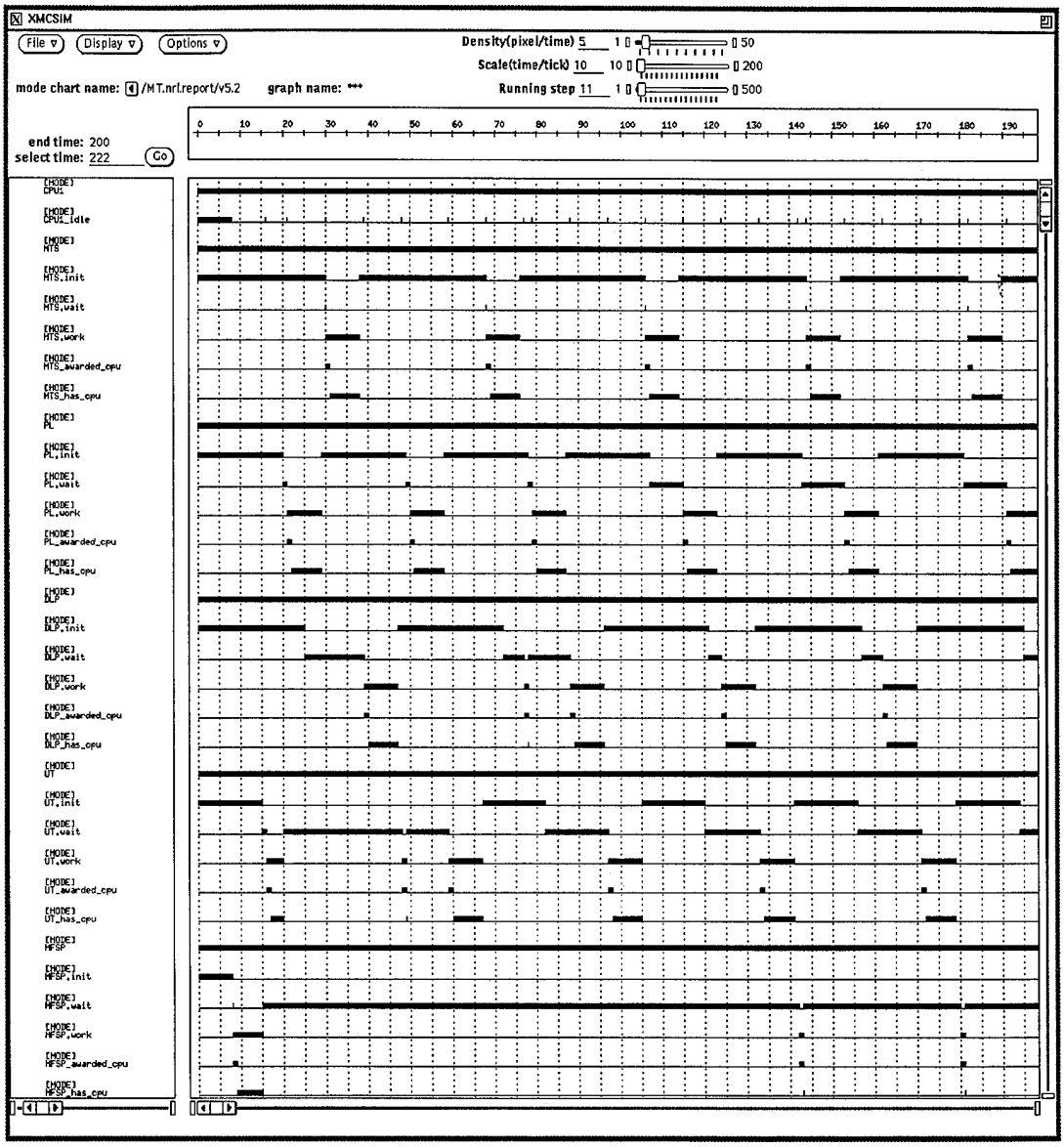


FIGURE 2. The Modechart Simulator.

The verifier is based on model-checking algorithms that compute the specifications entire computation graph and then traverse it checking for the states implied by each query. As such, it suffers from a state-explosion problem that renders it impractical for most real-world problems. However, state-compression techniques are being applied to ameliorate this problem, as is an approach that combines the verifier with the simulator, so that the combined analysis tool can simulate a single computation path up to a point, and then verify a sub-tree of the computation graph from then on. One can also imagine incorporating theorem provers into this framework.

Summary

As indicated previously, we have presented Modechart not because it is an ideal ADL, but because it is a language that can be used as an ADL and has an exemplary analysis environment. Modechart has been used to model the software of an air-to-air radar-guided missile, and the specification was primarily architectural in nature (i.e., the modes represented software components more than computation states) as described here. The analysis performed with the simulator and verifier indicated their usefulness in a development environment, and suggest ways in which formal methods may be incorporated into the software architecture framework.

References

1. Clements, P., C. Heitmeyer, B. Labaw, and A.K. Mok, "Engineering CASE Tools to Support Formal Methods for Real-Time Software Development," *Proceedings, 5th International Workshop on Computer-Aided Software Engineering*, Montreal, 7-9 July 1992. Also available as U. S. Naval Research Laboratory Report 202 (HCI-92-042), February 1992.

2. Clements, P., C. Heitmeyer, B. Labaw, and A. Rose, *A Toolset for Specifying and Analyzing Real-Time Systems: Overview and Example*, U. S. Naval Research Laboratory Report 7405, October 1993.

3. Clements, P., C. Heitmeyer, B. Labaw, and A. Rose, "MT: A Toolset for Specifying and Analyzing Real-Time Systems", *Proceedings, 1993 Real-Time Systems Symposium,* Durham, NC, December 1993.

4. Heitmeyer, C. and B. Labaw, "Software Development for Hard Real-Time Systems," *Proceedings, 7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, VA, 10-11 May 1990.

5. Jahanian, F. and A.K. Mok, "Modechart: A Specification Language for Real-Time Systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, December 1994, pp. 933-947.

6. Jahanian, F. and D. Stuart, "A Method For Verifying Properties of Modechart Specifications," *Proceedings, 9th Real-Time Systems Symposium*, Huntsville, Alabama, 1988.

7. Mok, A.K., "SARTOR-a Design Environment for Real-Time Systems," *Proceedings, 9th IEEE COMPSAC*, pp. 174-181, Chicago, Ill., October 1985.

8. Stuart, D.A., "Implementing a Verifier for Real-Time Systems," *Proceedings, 1990 Real-Time Systems Symposium*, pp. 62-71, Orlando, FL, 5-7 December 1990.

Lightweight Formal Methods

Dave Robertson
Department of Artificial Intelligence,
University of Edinburgh, Scotland
Email: dr@aisb.ed.ac.uk

Abstract

Formal methods are frequently presented as "heavyweight" solutions to the problem of high level specification. Users of such methods are expected to develop the types of intellectual "muscle" needed to understand a method in depth and, after intensive training, are then able to apply it to a wide range of problems. Whilst this ideal is worthy of support within formal methods communities, it seems unrealistic to expect those outside to accept such a heavy commitment of time and effort - particularly since many people with a stake in high level specification are not interested in mathematics or computer science. There is a need for "lightweight" formal methods, which can easily be picked up and which offer an obvious gain to practitioners after a short training span. This paper describes examples of lightweight tools, each of which relies on an architectural description of specifications which would not normally be apparent from examining the specification alone.

1 Introduction

In general terms, I think of an architecture as a means of conveying a standard style of construction and of use. Research in computational logic has produced a plethora of representational schemes and manipulation methods which are capable of assisting in the design of high level specifications. However, the academic success of these results has not been matched by their uptake in domains of application. One reason for this might be that the architectural features which help us judge whether a particular style of specification is relevant and useful are normally absent from formal theory. The problem is exacerbated by the natural urge to push the capabilities of our systems as far as they will go technologically and, in so doing, to distance them from all but a select group of users. Sometimes this can obscure a straightforward and effective technique which, used in an appropriate style, could have worked in a limited area of specification. Over the last few months I have been revisiting some of the existing work conducted within my own (and related) groups and attempting to identify core techniques which: make a useful contribution to specification design in a limited domain; can easily be communicated to users in that domain with small amount of training; and which can be implemented using a small amount of code. These techniques are collected within a single system (called LSS) but each can be operated independently. The only connection between them is that each produces specifications (or partial specifications) in Horn clause logic, which acts as a *lingua franca* between techniques. Some of the techniques currently implemented within LSS are described below. These involve: use of specification repositories via a "techniques editor" (§ 2); infiltration of a formal representation within an established informal method (§ 3); and provision of a high level graphical language for a subset of Horn clauses (§ 4). The implementations of these techniques make heavy use of graphics and windowing but space limitations make it impossible to present these here. Instead, a simplified sketch is given of some of the underlying formal mechanisms.

2 Standard Component Repositories

Some well established branches of formal specification have developed standard forms of description which are reused across applications. This is particularly true of the logic programming community, in which the same basic flows of control are used as the basis for a wide variety of programs. A number of research centres have taken advantage of this in order to provide support in the synthesis and transformation of specifications (for example, [4], [3], [1]). One way in which part of this research can be used to lighten the load of specification developers is to provide repositories of standard flows of control which can form starting points for elaboration into completed specifications. For example, the following specification for the predicate

$solve/1$ defines a flow of control for a simple meta-interpreter, of the kind given in most general-purpose logic programming texts. The predicate $base(X)$ defines the base cases for goal X; the predicate $implies(X, P)$ denotes that goal X can be solved if goal P can be solved; $(A \wedge B)$ is the conjunction of goals A and B; and $(A \vee B)$ is the disjunction of goals A and B.

$$\begin{aligned} solve(X) &\leftarrow base(X) \\ solve((A \wedge B)) &\leftarrow solve(A) \wedge solve(B) \\ solve((A \vee B)) &\leftarrow solve(A) \\ solve((A \vee B)) &\leftarrow solve(B) \\ solve(X) &\leftarrow implies(X, P) \wedge solve(P) \end{aligned}$$

Basic flows of control like the one above can be extended by applying a sequence of template transformations to the entire definition. The most interesting of these are transformations which add additional arguments to the predicate. For example, we might want to add to $solve/1$ an argument to record the proof tree obtained in finding a solution. This can be added in two stages. First we select and apply the general template for accumulators which requires the following transformation to each clause of the form $P(A_1, \cdots, A_n) \leftarrow Body$:

- Rewrite $P(A_1, \cdots, A_n)$ to $P(A_1, \cdots, A_n, R)$
- For each i'th recursive subgoal in $Body$ of the form $P(B_1, \cdots, B_n)$, rewrite it to $P(B_1, \cdots, B_n, R_i)$.
- Add to $Body$ the extra conjunct $\mathcal{F}(Vars, R)$, where $Vars$ is the set of all variables (excluding R) appearing in the clause. This represents a "gap" for a function, which must subsequently be defined, which must determine R from some subset of $Vars$.

Applying this to our example gives the following new definition for $solve/2$ in which the structure added by the transformation has been underlined.

$$\begin{aligned} solve(X, \underline{R}) &\leftarrow base(X) \wedge \underline{\mathcal{F}_1(\{X\}, R)} \\ solve((A \wedge B), \underline{R}) &\leftarrow solve(A, \underline{R_a}) \wedge solve(B, \underline{R_b}) \wedge \underline{\mathcal{F}_2(\{A, B, R_a, R_b\}, R)} \\ solve((A \vee B), \underline{R}) &\leftarrow solve(A, \underline{R_a}) \wedge \underline{\mathcal{F}_3(\{A, R_a\}, R)} \\ solve((A \vee B), \underline{R}) &\leftarrow solve(B, \underline{R_b}) \wedge \underline{\mathcal{F}_4(\{B, R_b\}, R)} \\ solve(X, \underline{R}) &\leftarrow implies(X, P) \wedge solve(P, \underline{R_p}) \wedge \underline{\mathcal{F}_5(\{X, P, R_p\}, R)} \end{aligned}$$

The second stage is to substitute for each $\mathcal{F}(Vars, R)$ a concrete function which will obtain R from $Vars$. Since we want to construct a proof tree each of these will correspond to unifications of R with the appropriate structured term describing the tree structure. Suppose that we choose the following replacements.

- $\mathcal{F}_1(\{X\}, R)$ replaced by $R = leaf(X)$
- $\mathcal{F}_2(\{A, B, R_a, R_b\}, R)$ replaced by $R = and(A, B, R_a, R_b)$
- $\mathcal{F}_3(\{A, R_a\}, R)$ replaced by $R = or((A, B), R_a)$
- $\mathcal{F}_4(\{B, R_b\}, R)$ replaced by $R = or(A, B), R_b)$
- $\mathcal{F}_5(\{X, P, R_p\}, R)$ replaced by $R = imp(X, P, R_p)$

Applying these and replacing explicit unifications in the normal way gives us the following specification which we could further extend by applying more templates.

$$\begin{aligned} solve(X, leaf(X)) &\leftarrow base(X) \\ solve((A \wedge B), and(A, B, R_a, R_b)) &\leftarrow solve(A, R_a) \wedge solve(B, R_b) \\ solve((A \vee B), or((A, B), R_a)) &\leftarrow solve(A, R_a) \\ solve((A \vee B), or(A, B), R_b)) &\leftarrow solve(B, R_b) \\ solve(X, imp(X, P, R_p)) &\leftarrow implies(X, P) \wedge solve(P, R_p) \end{aligned}$$

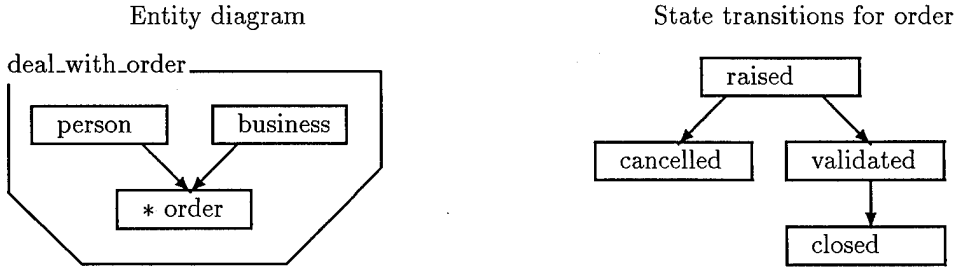
Who is this for ? People who are familiar with logic programming and who want rapidly to construct specifications by reusing established styles of construction.

Why is it lightweight ? The notation used throughout the system is familiar to its target users. The main overheads of adopting the system are in adjusting to the requirement of selecting a flow of control in the early stages of development, and in becoming familiar with the repository of program structures. In return for this adjustment, users gain the ability to create large portions of their specifications very quickly; to obtain without effort annotations describing key properties which their specification should preserve[1]; and to retain, behind the scenes, a detailed record of the main stages in construction.

What are its limitations ? This method is only as good as its repository and there is a conflict between the desire for a large repository of detailed examples (which is saves construction effort but makes it harder to find the best starting point) and a small collection of more general examples (which are easier to find but need more manipulation). The current editor provides general flows of control but allows users to employ these as patterns for searching existing files of specifications for structures which are syntactically similar. This allows limited reuse of existing specifications and relieves some of the pressure on the repository.

3 Following Established Methods

Many informal methods for high level specification are in use commercially. These are often fine-tuned to appeal to a target audience and (since they don't rely on formal theory to constrain their use) they often benefit from careful presentation and organisational support. One way of getting formal methods into use "by the back door" is to infiltrate them into informal methods without disturbing the (seeming) informality of the method. The example currently supported in LSS is IBM's Business System Development Method, which is used commercially to prompt business users for high level descriptions of the main components and interactions of their businesses. In previous work, we have produced a support system for BSDM analysts ([2]) and a lightweight version of this is being adapted for LSS. BSDM analysis begins by assisting clients to construct business maps, a simple example of which is given on the left of the diagram below. In BSDM terminology, three entities (*person*, *business* and *order*) are involved in a process called *deal_with_order*. The links between entities denote dependencies - thus an order cannot exist without an associated person and business. The asterisk next to *order* denotes that it is a focal element (an entity altered by a process). The business map is used to lead into the definition of state transitions for entities - the transitions for *order* are shown on the right of the diagram below. Attributes pertinent to the entities are also defined (for example an order might have the attribute *checked* which is set to *true* when it has been confirmed as valid).



This framework is used to prompt for sketches of the processes involved. The main components of this stage are triggering conditions and actions. For example, we might define a request from a customer as a triggering condition for initiation of the *raised* state in *deal_with_order*. Moving from the *raised* state to the *validated* state might involve setting the *checked* attribute to *true*. LSS stays close to the descriptive methods of the normal BSDM method so, as far as users are concerned, it acts simply as a note taking

[1] Space limitations prevent the description of this facility in the current paper

system. However, it converts this information into an executable form behind the scenes. Our running example would yield the transition descriptions shown below:

$$\begin{aligned} transition(\mathcal{S}, \mathcal{S}_2) \quad &\leftarrow \quad remove(customer\_request(person(P), business(B)), \mathcal{S}, \mathcal{S}_1) \wedge \\ &\qquad add(raised(order(P, B)), \mathcal{S}_1, \mathcal{S}_2) \\ transition(\mathcal{S}, \mathcal{S}_3) \quad &\leftarrow \quad remove(raised(order(P, B)), \mathcal{S}, \mathcal{S}_1) \wedge \\ &\qquad add(validated(order(P, B)), \mathcal{S}_1, \mathcal{S}_2) \wedge \\ &\qquad add(checked(order(P, B), true), \mathcal{S}_2, \mathcal{S}_3) \end{aligned}$$

A simple state transition meta-interpreter can then be used to generate possible states of the business model from these formal descriptions.

Who is this for ? People who are already promoting BSDM, normally by running organisational analyses, and who do not want to become heavily involved in formal methods but do want some of the benefits which formality provides (such as executability).

Why is it lightweight ? Since the formal system is non-intrusive there is little immediate cost to BSDM users in adopting it, beyond some adjustment to the layout of the computer system (BSDM is normally a paper based method). The formal notation in this case is firmly in second place to the larger scale method so it is regarded by our BSDM collaborators as just another tool for their analysis.

What are its limitations ? This approach is effective only for people who are trained in BSDM and to provide automatic translations from the diagrams we must make strong limiting assumptions about how the business model will be represented. Thus our easy route to acceptability of this formal approach is taken at the expense of a tightly restricted class of users.

4 Discovering Accessible Classes of Expression

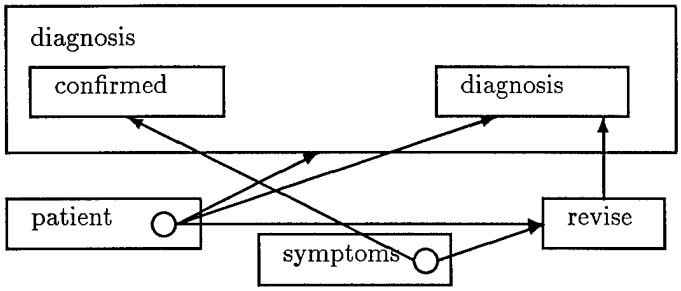
A common form of recursive structure in logic programming involves the use of a paired accumulator in order to construct a final result from intermediate results found on the way through a recursion. Consider, for example, the following specification for a simple diagnostic system, in which symptoms (S) for patient (P) are revised and hypotheses are updated recursively until a confirmed diagnosis is obtained:

$$\begin{aligned} diagnosis(P, S, \underline{R_s, R_s}) \quad &\leftarrow \quad confirmed(S, R_s) \\ diagnosis(P, S, \underline{R_s, R_f}) \quad &\leftarrow \quad revise(P, S, Sr) \wedge \\ &\qquad \underline{update\_hypotheses(Sr, R_s, R_n)} \wedge \\ &\qquad diagnosis(P, Sr, \underline{R_n, R_f}) \end{aligned}$$

The parts which are underlined in the definitions above constitute the paired accumulator. The first argument in the pair (R_s) contains the intermediate result, obtained on the way down through the recursion. In the base case this is matched to the second argument, instantiating the final result. In the recursive case, we update the current hypotheses (R_s) based on a revised set of symptoms to obtain a new intermediate result (R_n) which is carried down through the recursion on place of R_s. This technique is standard in logic programming (in fact it appears as one of the transformations in the structure editor of § 2). However it is known as a difficult concept to present to those without experience in this type of definition. In the LSS system we have provided a graphical language which is specialised to these types of recursions. It does not describe a complete specification like the one above but gives a useful skeletal description suitable for further development by the structure editor.

The graphical system uses only three types of symbol: a box representing a set; a circle representing a subset of a given set; and an arc which indicates a dependency between sets/subsets. By putting a box inside

another we denote inclusion of the smaller within the larger. For paired accumulator programs this gives a compelling visual metaphor for recursion because the structure of the accumulator guarantees that any results obtained from the accumulator in the recursive subgoal will also appear as results for the goal which initiated that subgoal. Thus, recursive subgoals always appear as subsets of the main goal in our diagrams. The full diagram for *diagnosis* appears below.



This diagram can be translated automatically[2] to the following specification:

$$
\begin{aligned}
diagnosis(P, S, \underline{R_s, R_s}) &\leftarrow confirmed(S, R_s) \\
diagnosis(P, S, \underline{R_s, R_f}) &\leftarrow revise(P, S, Sr) \wedge \\
&\quad \underline{\mathcal{F}(\{P, S, Sr, R_s\}, R_n)} \wedge \\
&\quad diagnosis(P, Sr, \underline{R_n, R_f})
\end{aligned}
$$

This is identical to the one we are looking for, except that we leave open the means of revising the hypotheses - specifying only that some function $\mathcal{F}$ should exist (see § 2).

Who is this for ? People who want to supply high level frameworks for recursive predicates but do not want to learn all of the mechanics of specification in the formal language.

Why is it lightweight ? The graphical system employs graphical metaphors which reduce the task of specification (at this level) to a small number of comparatively simple decisions (which sets are contained within others and on which sets do others depend). However, we have too little empirical evidence to claim empirically (as we did for the BSDM notation) that this technique is easy to use in practice.

What are its limitations ? This approach is targeted at a particular class of accumulator based specification. It does not allow negation or case splits in clauses, although these could be added by the structure editor. In collaboration with Agusti and collegues at the Institute for Artificial Intelligence in Barcelona, we are experimenting with a more complex graphical language which accommodates full first order predicate calculus, but this is much less lightweight than the system described here. An example of the use of this more complex language appears in [5].

5 Conclusions

All of the tools described in this paper rely on a particular, strongly reinforced, style of specification architecture. The techniques editor of § 2 expresses this by making explicit the assemblies of arguments in predicates which are responsible for key behaviours during execution; the BSDM interface of § 3 employs a domain-specific form of communication to make a particular class of specifications conform to the language of users in a specific domain; and the graphical system of § 4 uses familiar graphical conventions to make architectural features such as functional dependency and recursion more apparent. LSS is not a uniform

[2] Some diagrams require user interaction to disambiguate the argument positions but usually these can be inferred from the argument types.

specification "environment". The individual techniques which it contains are directed at different groups of users and at different areas of high level specification. The use of Horn clause logic as a *lingua franca* for the system allows the outputs of some of the systems to be used as starting points for others but there is no requirement for users of each technique to take this into account unless they want to do so. Only some of the available techniques are described in this paper. LSS also provides a window-based dialogue generation system, allowing rapid prototyping of basic interface facilities ; a file manager for retrieving and storing specifications; and a system for summarising the large-scale structure of specifications. In future, this list of facilities will be extended to include similarly lightweight techniques for argumentation, refinement and analysis.

References

[1] A. W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. *International Journal of Human-Computer Studies*, 41(3):329–350, September 1994. Also as Research Paper 641, Dept of Artificial Intelligence, University of Edinburgh.

[2] Y. Chen-Burger, D. Robertson, J. Fraser, and C. Lissoni. Kbst: A support tool for business modelling in bsdm. In *Proceedings of BCSES-95*, Cambridge, England, in press. British Computer Society Specialist Group on Expert Systems, Cambridge University Press. Also available as D.A.I. Research paper 338.

[3] N.E. Fuchs and M.P. Fromherz. Schema-based transformations of logic programs. In T.P. Clement and K. Lau, editors, *Logic Program Synthesis and Transformation, Workshops in Computing (Proceedings LOPSTR-91.* Springer Verlag, 1992.

[4] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for prolog programming. Tr 89-170, Case Western Reserve University, 1989.

[5] D. Robertson, N.S. Park, and J. Agusti. Layered design of kbs from specification to hardware. In *Proceedings of ECAI workshop on formal specification of knowledge-based systems*, Amsterdam, Netherlands, 1994.

THE SOFTWARE ARCHITECTURE FOR THE ANALYSIS OF GEOGRAPHIC AND REMOTELY SENSED DATA

by
Daniel E. Cooke and Scott A. Starks
Computer Science Department and the NASA Pan American Center for Earth and Environmental Studies
U.T. El Paso
El Paso, TX 79968-0518

1. BACKGROUND

Many federal agencies acquire, store, and process geographical data (which is typically acquired through ground studies) and image processing data (which is typically acquired through remote sensing). A significant amount of remotely sensed data is acquired through satellite observations. Software systems exist which provide for spatial and temporal query access to information about various regions of the earth. For example, from a low resolution map of the U.S. a user might outline an area such as Texas via a mouse and quickly view a map with much greater detail about Texas. In fact, the user might select from a number of possible views of Texas, including views which show roadways; vegetation; bodies of water and rivers; cities and towns; or any combination of these features and others not mentioned. From this standpoint the user could select a city in order to gather even more detailed information which might include streets, buildings, vegetation, fire hydrants, utility information (both above and below ground), population data, weather data, etc. Some of this data might come from remote sensing (for example buildings and streets could be obtained from visible data and vegetation from near-infrared data) and some data might be obtained from ground studies including the population and weather data. The user might also be able to obtain this same data along a temporal dimension. The remote sensing data is typically stored in a manner such that it is called *image data* or *raster data*, while the ground study data is called *vector data*. The vector data is typically associated with some *spatial key* and, from the database standpoint, might best be viewed as a tuple of data associated with some location.

The ability to view information as was just described requires a sophisticated software architecture (we will ignore the obvious hardware requirements). Figure 1 provides an overview of this architecture. The data acquired from remote sensing and ground studies must be preprocessed in order to make it suitable for display. Scientists develop exploratory programs which process the raw data, extracting those subsets of data which are of interest in some problem analysis. Image processing software is employed by analysts in order to develop displays of images which contain as much information content as possible. Once the data and images are of a sufficient quality they migrate into databases to be accessed, usually over networked systems, and displayed using a Geographical Information System (GIS). GIS systems typically provide access to all vector data and to some subset of image data. Some image processing information, however, cannot be combined with related GIS information. Clearly, the software architecture to support the analysis of earth data involves the use of a wide range of sophisticated software tools and exploratory programming languages.

There are two major problems with the current software architecture. The first problem is that it takes a significant effort to do the exploratory programming to extract data of interest from the raw data database. Furthermore, once the data has been processed and is residing in the processed data database, it takes too long to do further processing in a database host language or even in a procedural version of *SQL*. The second problem is that it is difficult to know how the high level tools will interact. In other words, there is no known formal specification language available to describe the functionality of the various tools which make up the architecture. If there were such a language and, for example, both the geographical information system and the database management system were specified in this language, the combined behavior of the two specified systems could be studied prior to attempts to put the systems together.

Recently, we were awarded a NASA Center aligned with the Mission to Planet Earth Enterprise. Half of the center's research activity is to be devoted to studying and trying to solve, among others, the two problems presented above. This paper provides an overview of what we intend to do.

2. THE NEED FOR DIFFERENT PROGRAMMING LANGUAGES.

It is estimated that less than 3% of the Satellite Data available has ever been seen. Less than 1% has been analyzed. In other words, we have the technical know-how to acquire and store the satellite data but we have difficulty determining the information content contained therein. Every year that passes adds more data to the existing unanalyzed data sets. The central problem has to do with programming. It seems that much of the exploratory programming to analyze the raw data sets is done in languages like FORTRAN or C. The resulting programming effort costs too much money and takes too much time for it to keep up with the growth in new data.

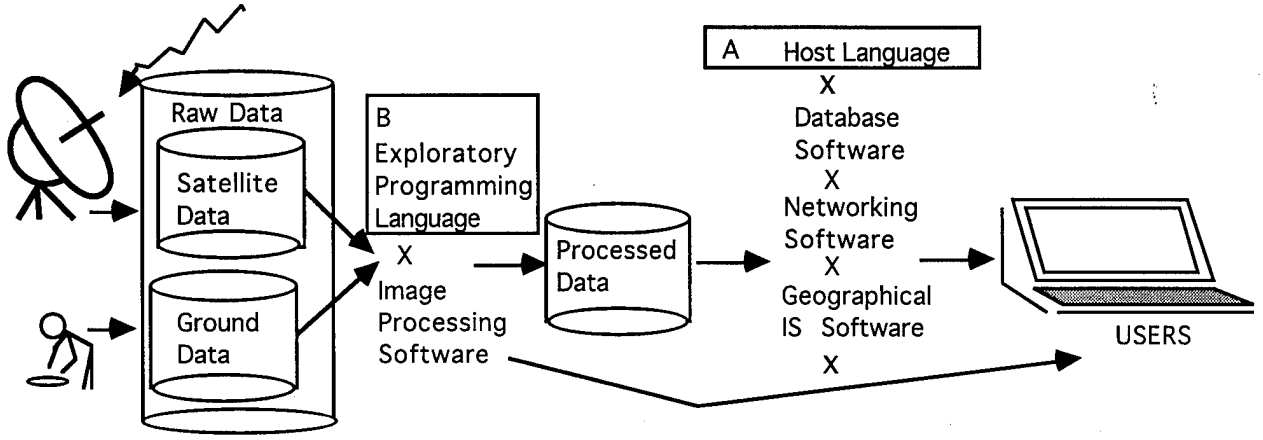


Figure 1. Software Architecture for Geographical Data Analysis.

SequenceL (formerly called BagL) is a language designed to experiment with a strategy for problem solving wherein one solves a problem by describing data structures strictly in terms of their form and content, rather than also having to describe algorithms to produce and/or process the data structures. The *SequenceL* work has led to the identification of constructs for describing data structures. Our view of a data structure is meant to include traditional data structures, databases, screen displays, reports, etc. The primitive data structure of *SequenceL* is the ***sequence***.

Sequences in *SequenceL* are collections of elements wherein each element may occur more than one time (as opposed to a mathematical *set*) and where each occurrence of an element possesses an ordinal position (as opposed to a *bag* or *multiset*). A ***sequence*** may be singleton (e.g., [99]), or nonsingleton (e.g., [[1],[2],[3]] or [[[1],[2],[3]],[[10],[20],[30]]). Complex structures of ***sequences*** containing ***sequences*** can be described. Like the list of LISP and the array of APL and J, the ***sequence*** of *SequenceL* can be used to build any data structure. We call the various data structures that can be composed using sequences, **nonscalars**.

It is believed that processing geographical data utilizing nonscalar processing constructs will improve the productivity in the exploratory and database programming efforts depicted in Figure 1, boxes A and B. This is our niche. Nonscalar processing constructs do not require the explicit use of iterative, recursive, or I/O constructs. Furthermore, early analysis of *SequenceL* seems to indicate that the nonscalar constructs provide a uniform abstraction for data definition, query, host language, and integrity constraint processing. Much effort is needed, however, to develop a robust, easy-to-use, production-quality language based solely upon these constructs. The production quality version of such a language must enhance our ability to develop large programs, particularly the types of programs which comprise the result of exploratory programming efforts.

2.1 Nonscalar Primitives.

A computational step, in *SequenceL*, involves the evaluation of a guarded command, which is applied to an input ***sequence***, and which yields an output ***sequence***. Functions contain the guarded commands of a *SequenceL* program. The inputs to a computational step are obtained in one of two ways: (a). through an **eventive** construct, wherein a function consumes its input from a universe of named sequences and produces, as its output, a set of named sequences (in the universe); or (b). through normal functional parameter passing, which is similar to the parameter passing in languages like LISP.

A total computation, in *SequenceL*, yields a partially ordered set of named ***sequences***. A command, in a *SequenceL* guarded command, falls into one of three categories: **generative**, **regular**, or **irregular**. These three

categories, combined with the **eventive** category, comprise the four constructs for nonscalar processing, and are used in place of algorithms.

There are two major concerns which arise when processing nonscalars: the selection of items to participate in the processing and whether processing reduces or expands input when producing output.

In nonscalar processing, a nonscalar (i.e., the *DOMAIN* column in figure 2) serves as the basis for a new nonscalar (i.e., the *RANGE* column in figure 2). A given construct acts in a manner to produce a *range* nonscalar based upon the *domain* nonscalar. All elements of a domain may participate in the computation needed to produce the range. The ***regular*** and ***generative*** constructs use all elements of a domain ***sequence*** in order to produce a range ***sequence***. For example, when one computes the *sum* of a set of elements, the addition operator is being applied, in a **regular** fashion, to *all* elements of the set (i.e., the addition operator is acting as an aggregate operator in this case). Alternatively, some subset of the elements in the domain ***sequence*** are used to produce the range sequence. The ***irregular*** construct selects items to participate based upon position or value, while the ***eventive*** selects elements based upon availability and (optionally) upon additional conditions such as position or value of the available domain element.

In producing a range sequence, nonscalar constructs either reduce or expand the domain ***sequence***. Reduction and expansion can occur with respect to the cardinality or dimensionality of the domain sequence.[1] Expansion means that the range ***sequence*** is larger, in dimension or cardinality, than the domain ***sequence*** upon which it is based. Reduction means that the range ***sequence*** is smaller, in dimension or cardinality, than the domain ***sequence*** upon which it is based. Both the reduction and the expansion may sometimes leave the range ***sequence*** equal, in dimension and/or cardinality, to the domain ***sequence*** .

In both the regular and irregular processing of nonscalars, one begins with a nonscalar and (typically) *reduces in the dimension* of the nonscalar (in the case of regular processing) or *reduces in cardinality* (in the case of irregular processing). The ***generative*** construct expands results.

	DOMAIN	**RANGE**	
	Selection	Dimension	Cardinality
Regular	All	Reduces	
Irregular	Some based on selection		Reduces
Generative	All	Expands	Expands
Eventive	Some based on availability	Reduce/Expand	Reduce/Expand

Figure 2. A Summary of Nonscalar Constructs.

2.2 Examples of Nonscalar Primitives in *SequenceL*.

The ***regular*** form of processing is used when an operation is to be applied in a uniform manner to *all* of the elements of the nonscalar. For example, when one computes the *sum* of a set of elements, the addition operator is being applied to *all* elements of the set (i.e., the addition operator is acting as an aggregate operator in this case). This construct typically reduces a structure in terms of its dimension. The example below reduces from a two dimensional sequence into a one dimensional:

$$*([[1],[2],[3],[4]]) \gg^{2} [(1*(2*(3*4)))] \gg \gg [24] \qquad (1)$$

In other cases, one may wish to apply an operator to *some* of the elements of the nonscalar. The selection of elements may be based upon *value* or *position*. *SequenceL* possesses a construct for this ***irregular*** form of processing data. The ***irregular*** construct requires the ability to select items conditionally based upon a *map* or subscript. In *SequenceL* this construct requires a guarded command structure. The noniterative form of the guarded command provides the basic form of the body of a *SequenceL* function. For example, suppose one wished to select all odd values in some sequence of integers, *int*. To do so, one would need to provide a function:

---

[1] A reducing construct, given an input D, will produce a result R where R is no larger (in dimension or cardinality) than D. E.G., Given a construct that reduces in cardinality: $D \rightarrow R$, the following relation holds after D is mapped to R: $|D| \geq |R|$

[2] This operator represents a *SequenceL* evaluation step. The use of » » indicates many *SequenceL* steps.

Odd_Integers(Domain(Int),Range()) =
[[Int(i)] when ≠(mod(Int(i),2),0)
[] otherwise]

This function effectively states that one wishes to return each *ith* value of *Int* when that *ith* value is odd (i.e., when the *ith* value *mod 2* is not equal to zero). *Odd_Integers* has no *range* arguments because it is intended solely for use as a helper function. That is, it is to be invoked by some other *SequenceL* function which will supply its argument *Int* and accept its result. Suppose one wished to sum all odd integers in *Int.* To do so, one would compose functions:

Sum_of_Odds(Domain(Int),Range(Sum)) =
[+(Odd_Integers(Int))]

This function is considered to be a nonhelper function.

A *SequenceL* program consists solely of a set of *SequenceL* functions which are applied to a *SequenceL* **Universe**, *U.* The **Universe** is where the user may pair *SequenceL* variable names with ***sequences*** of values via a text editor. Like GAMMA [B93, HLS92] (and similar to the tuple-space of **Linda** [G85]), when a *SequenceL* program executes it modifies the **Universe** to which it is applied. If each variable appearing in a function's *Domain* is paired with a ***sequence*** in *U* , the function is enabled for execution and is termed a **nonhelper function**. Obviously, several functions may be thus enabled. These functions execute concurrently. When a nonhelper function executes, it consumes all variables in its domain. When a nonhelper function completes execution, the result(s) of the function is(are) paired with the function's range variable(s) and added (or produced) in the **Universe**. It is via the **Universe** that a user provides inputs and obtains outputs. There are no explicit constructs for I/O in *SequenceL.*

The interaction between a *SequenceL* function and its **Universe**, in terms of production and consumption, provides for the ***eventive processing of a nonscalar structure.*** In eventive processing, one processes a nonscalar structure whose elements are not necessarily available all at once. The arrival of an element is an event to which a function must respond.

Assume program $\prod$ is applied to universe *U,* where $\prod$ contains the functions *Odd_Integers* and *Sum_of_Odds,* as introduced above, and *U = { <Int, [[6],[7],[8],[3]]> }.* Based upon the *eventive construct,* only *Sum_of_Odds* is enabled for execution. It consumes its arguments from *U:*

U = { }

and immediately invokes *Odd_Integers* which is applied to each value of *Int. Odd_Integers* returns only those values which meet the *≠(mod(Int(i),2),0)* condition. Since, empty sequences *[]* are null values they disappear in the result of *Odd_Integers* which is returned to *Sum_of_Odds:*

Sum_of_Odds(Domain(Int),Range(Sum)) = [+([[7],[3]])]

Notice that *Odd_Integers,* which is an example of irregular processing, reduces in cardinality. Since *Sum_of_Odds* is a nonhelper function, its result is computed, paired with the respective range variable and *produced* in the universe:

U = { <Sum, [10]> }

The eventive construct also allows for the processing of *integrity constraints.* Rather than enabling nonhelper functions based upon the availability of domain variables in the universe, constraint functions are enabled based upon the availability of a universe *U* and its successor *U'.* Thus, transition constraints can be stated exactly one time and imposed automatically whenever there is a change to a universe.

In both the regular and irregular processing of nonscalars, one begins with a nonscalar and (typically) *reduces in the dimension* of the nonscalar (in the case of regular processing) or *reduces in cardinality* (in the case of irregular processing). There are times, when it is necessary to *expand* a nonscalar, increasing the nonscalar in dimension and/or cardinality. *SequenceL* possesses a ***generative*** construct which is used to expand nonscalars. Its basic form is to provide an upper and lower boundary for the term to be generated and an optional membership condition for items to be generated. For example, the term *[[1],...,[5]]* evaluates to *[[1],[2],[3],[4],[5]]* in *SequenceL*

while *[[0],[1],...,=([!,+([pred(pred(!)),pred(!)])]),...,[20]]* evaluates to *[0],[1],[1],[2],[3],[5],[8],[13]].* It seems clear, that there are four ways to expand:

From left to right:				
(lower bound)	→	(membership condition)	→	(upper bound)
From right to left:				
(lower bound)	←	(membership condition)	←	(upper bound)
From outside-in:		(e.g., convergence problems)		
(lower bound)	→	(membership condition)	←	(upper bound)
From inside-out:				
(lower bound)	←	(membership condition)	→	(upper bound)

The denotational semantics of *SequenceL* were completed in 1993. A prototype interpreter was completed in 1994. A proof that *SequenceL* is equivalent to the Universal Turing Machine was completed early in 1995.[F95] Many of these results are reported in earlier papers [C90-94, CG91]

3. SPECIFYING INTEGRATED SOFTWARE.

The second problem facing our center has to do with the integration of off-the-shelf tools and exploratory languages to form an appropriate software architecture for our mission. Within the next eighteen months we will acquire and/or develop software to provide the software architecture presented in Figure 1. Excellent off-the-shelf software exists to provide the image processing, networking, database, and geographical information system functionalities. The concern that we, and many others, face is our level of ignorance about the combined behaviors of resulting integrated systems. We are forced to (a). rely on the experiences of others who have combined some subset of these systems already or (b). put together software products and then stand back and observe what happens.

Facing this prospect has caused us to pause and wonder about the feasibility of identifying or developing a single specification language that could be used to describe the functional behavior of existing software products. A language such as this would need to have a precise semantic and be reasonably compact and easy to use. Based upon the language, it might be possible to develop an automated tool that could help one analyze the likely result of combining different software packages. Consider the following example.

Suppose one wished to combine a very simple relational database management system with a networking system. Assume that the database management system's interface is limited to a simple query language consisting of the relational algebra primitives: *union, difference, cartesian product, select,* and *join.* Assume that the network package consists of transaction primitives *post* and *receive.* Now imagine that both sets of transaction primitives have been defined precisely in some specification language. As an exercise, one could easily do this in Prolog. One can imagine the following combinations:

NETWORK	X	DATABASE
post		*difference*
post		*union*
post		*cartesian product*
receive		*select*
receive		*join*

Furthermore, one can imagine the combination of database transactions. For example, one might wish to select some set of tuples resulting from a cartesian product via the network (i.e., an interaction of *network x (database x database)*). In any case, if a simulator based upon the specification language were available, it might be possible to study the functionality that results from combining these two products. Furthermore, analysis tools could be developed to study the interactions as well. For example, one might wish to study the problems that arise with multiple users (e.g., *race* and *deadlock*), or those that occur due to network or database failures. The basic idea is that the specifications of the products could be studied at a fine level of detail to provide much greater predictability to the actual integration of tools.

The long term goal of a research effort to develop a specification language for the integration of tools would likely be to move the state of the practice towards a standard language. Suppose, in addition to receiving the normal set of brochures concerning a software product, one could *ftp* a specification of the product in this standard language. We intend to investigate the use of languages such as PSDL [BL90], MPL [W92], and RAPIDE [LK95] for specifying the functionality of tools identified in Figure 1.

Obviously the identification or development of such a language is a daunting task. The array of high level software tools currently available is quite large. Many of these tools have hundreds of commands. For example, in most image processing packages there exist many commands that can be used in an orthogonal way - there are commands to allow for data retrieval, processing, refining, etc.

Many areas of software engineering possess expertise to contribute to the endeavor introduced above. The areas include software slicing and merging, specification languages, reverse engineering, integration, etc. We believe that this area of software integration is where many computer scientists and software engineers will work in the future. We further believe that this represents a fundamental area of research, possessing a significant practical import.

4. CONCLUSIONS.

It is our belief that the "reverse" prototyping of software packages is a promising approach to the the analysis of how packages will behave in the context of large software architectures. In addition to the *SequenceL* language development activity, our center intends to use its software architecture as a laboratory for the study of methods to analyze software architectures. Our initial investigations will involve the use of the CAPS prototyping tool. In the initial invetigations we intend to prototype the various software packages we purchase. We will then combine the prototypes of the packages to see what functionality ensues. Since we are likely to purchase more than one GIS and more than one image processing package, we should be capable of determining the relative merits of the competing packages with respect to the total functionality of our software architecture.

5. BIBLIOGRAPHY.

[B93] J.P. Banatre and D. Le Matayer, "Programming by Multiset Transformation," *CACM,* Vol. 36 No. 1, (January, 1993), pp.98-111.

[BL90] V. Berzins and Luqi, "Languages for Specification, Design, and Prototyping", in P. Ng and R. Yeh (eds.), *Modern Software Engineering Foundations and Current Perspectives*, Van Nostrand Reinhold, pp. 83-118 (1990).

[C90] D.E. Cooke, "Towards a Formalism To Produce a Programmer Assistant CASE Tool," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2 No. 3, September, 1990, pp. 320-326.

[CG91] D.E. Cooke and A. Gates, "On the Development of a Method to Synthesize Programs from Requirement Specifications," *International Journal on Software Engineering and Knowledge Engineering*, Vol 1 No 1, (March, 1991) pp. 21-38.

[C92] Daniel E. Cooke, "An Issue of the Next Generation of Problem Solving Environments," *Journal of Systems Integration*, Vol 1(2), (February, 1992) pp. 39-52.

[C93] Daniel E. Cooke, "Possible Effects of the Next Generation Programming Language on the Software Process Model," *International Journal on Software Engineering and Knowledge Engineering*, Vol 3 No 3, (September, 1993) pp. 383-399.

[C94] Daniel E. Cooke, "Towards a Formalism for Program Generation," for the Air Force Office of Scientific Research, #F49620-93-1-0152, February, 1994.

[F95] B. Friesen, "The Universality of BagL," Masters Thesis, University of Texas at El Paso, May, 1995.

[G85] D.Gelernter, "Generative Communications in Linda", *ACM Transactions on Programming Languages and Systems*, **7** (1), pp. 80-112 (1985).

[HLS92] Chris Hankin, Daniel Le Metayer, and David Sands, "A Calculus of Gamma Programs," Publication Interne no 674, Juillet, 1992, IRISA, France.

[LK95] D.C. Luckham, J.J. Kenney, et. al. "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering,* Vol. 21 No. 4, (April, 1995) pp. 336-355.

[W92] G. Wiederhold, P. Wegner, S. Ceri "Toward Megaprogramming," *CACM,* Vol. 35 No. 11, (November, 1992) pp. 89-99.

ACKNOWLEDGEMENTS

Research sponsored by the Air Force Office of Scientific Research (AFSC), under contract F49620-93-1-0152; NASA, under contract NAG 2-670 Supp. # 2, and by NSF grant No. CDA-9015006 and NASA NCCW-0089.

An Animation Tool for Supporting Specification-Based Software Architectures

Krzysztof Czarnecki[†] *and Du Zhang*
Department of Computer Science
California State University
Sacramento, CA 95819-6021
{czarneck, zhangd}@gaia.ecs.csus.edu

Kevin Lano
Imperial College
180 Queens Gate
London SW7 1LU, U.K.
kcl@doc.ic.ac.uk

Abstract. *One of the important issues in supporting specification-based software architectures is how to verify that formal specifications of a software system satisfy liveness, safety and timing constraints, and are consistent and complete with regard to user requirements. Animation has been proposed as one of the techniques for achieving this goal. This article presents a practical approach and a tool to animating object-oriented formal specifications written in the language Z++. Implemented in Prolog, the tool handles inheritance, object identity, invariant and precondition checks, both the logical and the sequential method composition, and nondeterminism. It offers a solution to the mutable state problem. The merits of applying the tool to validating formal specifications are demonstrated in a case study. Animation is shown to be helpful in gaining the insight on how the specified system works, in validating the conformance of the formal specification to particular user requirements, as well as in "debugging" the formal specification.*

1 Introduction

It is essential that formal specifications of a software architecture satisfy liveness, safety and timing constraints [19] and correctly and completely capture all customer requirements [9]. Unfortunately, this formal specification development process is inherently error-prone and cannot be completely automated or proved [12]. *Inference* and *animation* have been proposed as the two methods which can help validate the formal specification against the informal one. Inference is used to answer specific questions about the formal specification, thus allowing the specifier to check if certain properties of the specified system conform to the informal requirements. Animation is a way of "executing" or "walking through" a formal specification. It makes it possible to "run" (animate) the specification in different scenarios, thus offering a means of dynamically testing of formal specifications. For that reason, an animation tool is an indispensable component of any specification-centric software development environment.

The purpose of this article is two-fold: to explore an animation-based approach to developing correct specifications for specification-based software architectures, and to describe an animation tool, the Z++ Animator, which animates formal specifications written in the object-oriented language Z++. The Tool is designed as an integral part of the Z++ Toolset[1] [17] and is implemented in Prolog. It covers a substantial subset of Z++ as defined in [17] (in particular, object creation, message passing and method composition).

The rest of the paper is organized as follows: Section 2 briefly reviews the specification language Z++. Section 3 deals with issues pertaining to animation. Section 4 discusses the Z++ Animator. A case study is given in Section 5. Related work is provided in Section 6. Finally, Section 7 concludes the paper with remarks on future work.

2 Object-Oriented Specification Language Z++

The object-oriented specification language Z++ stems from the Esprit II project REDO [17]. It was developed in 1989 independently from other similar research (resulting in languages such as Object-Z or MooZ; for descriptions of object-oriented specification languages see [16,26]).

Z++ extends the specification language Z by allowing the definition of classes. The current version of Z++ also supports *object identity*, *history constraints*, and explicit *method preconditions*. For a comparison of Z++ with other object-oriented specification languages see [15].

Since "plain" Z has a well defined semantics [25], the semantics of Z++ is defined in terms of a translation into the "plain" Z. The model-based semantics for Z++ is defined in [17], and [14] investigates the model-based semantics for object-oriented extensions to Z in general. The model-based semantics for Z++ provided the basis for the implementation of the Z++ Animator.

---

† *current address: Daimler-Benz AG, Research and Technology Center, Ulm, Germany.*
czarnecki@dbag.ulm.DaimlerBenz.COM

[1] The Z++ Toolset contains facilities for refinement, semantic analysis (automatic generation of proof obligations based on the class consistency conditions), translation into Z (or "flattening"), B and Object-Z, and animation. It also has a generator for producing an internal representation of an SSADM data-flow diagram from a class definition. The entire toolset is written in Prolog and has the status of a prototype.

3 Approaches to Direct Animation

There are two primary types of animation: indirect and direct. Indirect animation is virtually equivalent to *prototyping*. It involves the implementation of a prototype (often in an imperative language). In case of direct animation, on the other hand, the original formal specification is usually translated into a declarative language automatically or directly interpreted. Direct animation is preferable in assessing the validity of a formal specification since it eliminates the possibility of errors that can occur during the implementation of a prototype.

Because specification-based architectures require nondeterminism in order to be able to describe a family of software systems, direct animation (from now on referred to shortly as animation) has to be able to handle nondeterminism contained in a specification. This requirement suggests the use of declarative programming techniques. There has been a considerable amount of research on animation of Z specifications utilizing functional or logic-oriented languages (e.g. Prolog [6,8,27], Lisp [22], Miranda [7], and SQL [18]). The following two sections describe and compare two alternative approaches to animation.

3.1 "Exhaustive" Animation

A Z schema is represented as a Prolog rule in [6]. This Prolog rule is defined with goals derived from the *schema declarations* acting as generators, and goals derived from *predicate parts* as tests. The *given sets* are instantiated with "representative elements" of the required type. This approach employs a "generate and test" cycle to calculate all possible solutions described by the rule (and so by the corresponding schema) by iterating through all the given sets. If the schema and the given sets have more than one set of satisfying variables, all these sets can be usually found through backtracking. This is the reason why we refer to this type of animation as "exhaustive". Authors of [6] note further that a direct translation into Prolog will usually execute very inefficiently. In [5], Dick and Krause describe a transformation system which performs unfolding, goal promotion/demotion, and removal of duplicate goals to limit the search space. Unfortunately, in case of more complicated specifications, this approach requires a set of very sophisticated rules and there are always some aspects not covered by this set.

3.2 Animation with "Scenarios"

A different solution to the aforementioned inefficiency problem has been proposed by West and Eaglestone [27]. They use given sets for checking if the values of variables are in their domains rather than for generating these values. All given sets and variables (including the output variables) have to be instantiated before the animation can take place. This avoids generating the powersets in the first place. In other words, this approach allows us to check particular cases (or "scenarios") instead of generating all possible solutions (as was required in the exhaustive approach). This approach has been adopted in our animation tool. In addition, the Z^{++} Animator is capable of handling nondeterminism as well as cases in which the tool does not have the applicable algorithm to compute a particular equation. In these cases, the tool will ask the user to provide the resulting state of a method call. This state will then be "tested", instead of being computed, to see if it is a valid solution of the method predicate.

3.3 Animation of Object-Oriented Formal Specifications

The animation techniques presented above can also be used for the animation of object-oriented specifications. However, extensions are needed to cover issues unique to object-oriented specification languages, such as *recording state changes*, *object identity*, *inheritance*, *invariant checks*, *history checks*, *method composition*, and *precondition checks*.

Recording state changes or the problem of the *mutable object state* (i.e. "changing state") is one of the most difficult ones. This problem results from the persistence of objects, which must exist beyond the scope of single method invocations. If we use the correspondence between a schema and a method then the method can be represented as a Prolog rule. In computational logic, the scope of a variable cannot go beyond that of a clause. This creates problems when the method changes the state of some objects --- these changes have to persist between single method invocations.

Another problem arises, if a method call fails. From the viewpoint of logic, a failure should trigger the abortion of this call and all subcalls. Abortion means that the state of the system as it was before the method call has to be reestablished (as opposed to reestablishing just the invariant as in case of a programming language, e.g. *organized panic* in Eiffel [21, page 249]).

The problem of mutable object state originally comes from the area of logic-based object-oriented programming. Whatever the solution to this problem is, it has to fit in the formal framework of the logic-oriented language. Several possible solutions to this problem are surveyed in [2].

The standard way of recording global state changes in Prolog is the use of *assert* and *retract*. These two

constructs have no proper axiomatic semantics in Prolog. In particular, a simple use of *assert* and *retract* causes problems on backtracking. Backtracking can uninstantiate a previously instantiated variable, but any state that has been asserted by *assert* will not be "undone" on backtracking, in general. One solution to this problem is the use of backtrackable versions of *assert* and *retract*. The current implementation of our animation tool uses backtrackable versions of *assert* and *retract* to record the changes of object state. The idea behind this particular implementation was taken from [23] and it can be explained using the following Prolog clauses:

```
backtrackable_assert(State) :-
   asserta(State).

backtrackable_assert(State) :-
   retract(State),
   !,
   fail.
```

A call to backtrackable_assert/1 will create a choice point to remember the second clause. On backtracking, the second clause will "undo" the changes made by the first one.

Object identity involves the use of unique identifiers to reference objects. Object identity is a key concept in the so called *reference semantics* (see [16, pages 46-47]). Under this semantics, passing an object as an argument of a method involves passing the corresponding identifier (i.e. there is no value copying). This can potentially cause "side effects". For example, one problem due to such side effects can occur in case of *object sharing*. Object sharing involves an object which is referenced from two or more other objects. An execution of an operation of one of the latter objects could affect the state of the aliased object. There is no guarantee that these state changes will preserve the invariant of the remaining objects. A "global" invariant check could be used to solve this problem.

Inheritance is a relatively easy concept to implement. Name clashes can occur in case of multiple inheritance.

Invariant and history checks have to be performed with each method execution. The invariant has to be reestablished after the method has been completed. The execution of a method can only take place if this is not prohibited by the RTL formula in the history part of the receiver class. The history checks are currently not implemented in the animation tool.

Method composition by logical connectives (*and*, *or*, etc.), and the parallel and sequential compositions have to be provided. Method composition allows us to define new methods using already defined ones. This enables a more compact hierarchical design and prevents unnecessary errors since the new methods can use the old reliable and tested code.

Logical connectives act quasi in parallel (this is sometimes referred to as *and-* and *or-parallelism*). In the case of the conjunction of two method calls, the second method call just further constrains the set of possible "target" states of the first one. In terms of the Prolog implementation, the second call will have to unify its decorated attributes with those of the first call. The undecorated attributes are the same for both calls.

After unfolding the method calls into "plain" Z (see [17, pages 6-8], or [14, pages 239-241] for guidelines), the semantics of the sequential composition is given by the definition of the sequential composition of two Z schemas. In the Prolog implementation, this means that the values of the undecorated attributes seen by second operation in the composition have to be equal to the values of the decorated attributes of the first operation.

The translation of a method call into a schema requires the inclusion of any other directly or indirectly made calls inside this method. Thus the composition affects not only the top-level calls, but also all the internal ones.

Precondition checks guard the execution of a method. If the precondition predicate reduces to true, the method will be executed, otherwise not. The precondition should specify the domain of the method inside which the method succeeds.

4 The Z++ Animator

The functionality covered by the Z++ Animator includes: (1) algebraic evaluation predicates; (2) single inheritance; (3) support of the self-reference; (4) logical and sequential method composition; (5) full invariant checks (i.e. including all inherited invariants); (6) method precondition checks. The Z++ Animator uses the same internal Prolog representation of the Z++ code as in the Z++ Toolset. For the following Z++ specification of a counter:

```
CLASS Counter
OWNS
 value : Z
INVARIANT
 true
RETURNS
 Value : → Z
OPERATIONS
 SetValue : Z → ;
 Incr : → ;
ACTIONS
 Value val! ==> val! = value;
 SetValue val? ==> value' = val?;
```

Incr ==> *value'* = *value* + 1;
END CLASS

the whole class definition *Counter* is represented as a Prolog fact "class_definition/6", as shown below. There is a Prolog-based translator between the LaTeX format and internal representation [4].

```
class_definition(
  'Counter',
  [],
  [[types,value,integer]],
  true,
  [['Value',[],[[output,val]],true,[eq,[output,val],value]]],
  [
    ['SetValue',[[input,val]],[],true,[eq,[next,value],
                                       [input,val]]],
    ['Incr',[],[],true,[eq,[next,value],[add,value,1]]]
  ]).
```

Method Execution. The whole animation process consists of subsequent method executions. A method execution corresponds to the evaluation of its predicate. Before a predicate is evaluated, the values of the actual input arguments and the current values of the attributes of the receiver object have to be substituted for the corresponding variables in the predicate. During the evaluation, all symbols are treated as Z values. If it is necessary to resolve the value of a particular symbol (with respect to the environment), the value can be substituted for this symbol before the operation.

Object Representation. Each object is represented by two Prolog facts, one containing the values of the attributes, and the other containing the values of the next attributes. An object (and so its corresponding facts) is identified using a unique identifier, called *oop* (object-oriented pointer), which is generated when an object is created. The facts also contain the name of the creator class (i.e. the class of the object) and a unique name by which the objects can be conveniently identified by the user (an object does not have to have a name, but it has to have an oop).

Inheritance. The current implementation supports only the single inheritance. There are two situations where the inheritance takes effect: object creation and a method call.

Method Composition. A very important issue in object-oriented specifications is the method composition. Method composition allows us to define new methods using already defined ones. The current implementation covers logical *and* and *or* as well as the sequential compositions.

Invariant and Precondition Checks. During the animation process the method preconditions and the object invariants have to be evaluated to make sure that the state of the system does not violate any of them. A precondition check is performed before a method is called. If the outcome is *true*, the method is executed. If the check fails, the method is not executed and the method call returns *false*. The invariant check is performed upon a successful completion of a method call. This is sufficient if the sequential composition is used to compose methods only.

Implementation of the Z++ Animator. The Z++ Animator is implemented in Quintus Prolog™[2]. It is contained in two Prolog modules of over 150 rules, which in turn utilize eight other modules (rewriting predicates, parser) from the Z++ Toolset of more than 1100 rules. The performance of the tool is satisfactory. During the animation of the case study specification, each trace was produced in a few seconds (excluding the waiting time when the user input is needed).

5 Case Study

A case study was conducted in order to show how the Z++ Animator can help in the process of validating a formal specification of a system against its informal specification. A Mobile Phone System (MPS), taken from [24][3], was used for this case study. Because of the size of the case study, it was not possible to present the formal specification and the traces in this article. For a detailed discussion, the reader may refer to [4].

A *requirements traceability matrix* [4] between the informal and formal specifications of the MPS has been derived. This matrix lists the correspondences between each informal requirement and parts of the formal specification which implement this requirement. In cases where this correspondence is not obvious or the requirement is not explicitly represented in the formal specification, testing scenarios are proposed. These scenarios can be used to verify that the corresponding requirement is indeed reflected in the formal specification.

The Z++ Animator was used to animate the test scenarios contained in the requirements traceability matrix as well as some representative scenarios showing the MPS in operation. These representative scenarios were animated in order to demonstrate the general idea of how the MPS works. These scenarios should preferably use all methods defined in the formal specification and be representative with regard to all capabilities of the system. We refer to this type of animation as *exploratory animation*.

The animation results showed that the formal specification satisfies all but one corresponding

[2] Quintus Prolog™ is a trademark of the Quintus Corporation.

[3] [22] contains a formal specification of the Mobile Phone System written in Object-Z. This specification has been translated into Z++ for the purpose of this case study.

informal requirements. The cause of this inconsistency was a faulty informal requirement which was inconsistent with other informal requirements.

6 Related Work

Whereas the animation of specifications formulated in non-object-oriented specification languages (see Section 3) has been described quite well in the literature, the area of animating object-oriented specifications is still a very new one. The major problem is that the object-oriented specification languages are still in development and not all parts of their semantics have yet been completely defined. There are at least three extensions of Z which have a tool-support for animation. These languages are

- MooZ [20]. ForMooZ is an environment which supports construction, management and prototyping of MooZ specifications. Prototyping is done by the automatic translation of specifications into Smalltalk. During this translation, nonexecutable parts of a specification are marked pending and have to be refined later. For this reason, ForMooZ supports only indirect animation.
- OOZE (Object-Oriented Z Environment) [1, page 191]. The tool is build on top of the OBJ3 [10] system. However, the interpreter animates specifications written only in an executable sublanguage. In other words, the schema has to explicitly define the new values of the attributes of the object (thus the specification cannot contain any nondeterminism).
- Object-Z [11]. In [11] Hasselbring presents an Object-Z specification of a language for prototyping parallel algorithms. He also shows how this specification can be animated using the set-oriented language ProSet by manually translating the specification into ProSet. However, his approach does not allow to express the full power of Object-Z in the ProSet. It is somewhat similar to the approach of translating Object-Z specification into C++ as proposed in [13]. Thus it can be seen as an indirect approach to animation.

It is the recent work reported in [14,17] that provided a foundation for the development of the Z++ Animator. A less restrictive approach to animation has been adopted in the Z++ Animator. There are no restrictions on the form of the method to be animated. If the tool is not able to compute the outcome of a method (because of nondeterminism, or lack of the required algebraic simplification algorithm) the user will be asked to provide the resulting state. This state will then be checked against the method predicate for validity. If the method fails for this state, alternative states can be tried out.

7 Conclusions

In implementing the Z++ Animator, we focus on the following two major issues:

- Handling the mutable state problem. The "backtrackable assert" encapsulates the fact of storing the entire history of an object, thus offering an elegant solution to the problem.
- Animating method compositions. The sequential method composition is especially needed for the formulation of test scenarios, whereas the logical method compositions make it possible for system specifications to be more compact.

The specification of the Mobile Phone System comprises almost eight pages of Z++ code. A specification of a "real-world" system could involve several hundreds pages of code; however, using even this short specification example, the work presented in [4] demonstrated several benefits of animation.

The "exploratory animation" helps to quickly gain insight on how the specified system works. This is the experience of the authors. The actual animation process, in which the user has to provide some extra input and also can observe its progress, seems to explain the system more efficiently than the analysis of the rather lengthy traces. However, a trace can be used as a documentation of how the system behaves with respect to a particular scenario.

The case study also shows how animation can be used in validation of formal specifications against the informal user requirements. The idea behind a formal specification is to directly and explicitly capture the informal requirements in the form of logical sentences. In cases where this correspondence cannot be established or the specification is inconsistent, the animation of certain test scenarios can help. The test scenarios can be formulated directly from the given requirement without any detailed knowledge about the intrinsics of the formal specification. The animation of these scenarios can reveal errors and increase our confidence in the specification. The types of errors which can be revealed through animation include: (1) inconsistencies between the formal specification and the informal requirements as well as inconsistencies within the formal specification or the informal requirement itself; (2) incompleteness of the formal specification (This amounts to showing that certain scenarios are not properly handled by the system); (3) unused methods (detected through the animation of representative scenarios); and (4) some potential "infinite loops".

Finally, our experience also shows the usefulness of the Z++ Animator as a debugging facility. This was the

case during the translation of the original specification of the Mobile Phone System from Object-Z into Z^{++} where the Z^{++} Animator helped to locate several errors in the Z^{++} specification. More systematic techniques for deriving test scenarios based on formal specifications are discussed in [3].

The future work on the animation tools includes: extending language coverage (e.g. multiple inheritance, history checks, spontaneous actions, and more algebraic evaluation predicates) as well as providing a better user interface (especially, source-code-level debugging in addition to traces).

References

1. Alencar, A.J., J. Goguen. OOZE: An Object Oriented Z Envionment. *Proc. of ECOOP'91*, Vol. 512 of *Lecture Notes in Computer Science*, pp.180-199, Springer Verlag, 1991.
2. Alexiev, V. Mutable Object State for Object-Oriented Logic Programming: A Survey. Technical Report TR 93-15, Department of Computing Science, University of Alberta, August 1993.
3. Carrington, D., P. Stocs. A tale of two paradigms: Formal methods and software testing. In J.P. Bowen, J.A. Hall, editors, *Z User Workshop*, Cambridge 1994. Proceedings of the Eight Z User Meeting, pp. 51-68, Springer Verlag, 1994. Also available as Technical Report 94-4, University of Queensland, Department of Computer Science, Software Verification Research Centre, February 1994.
4. Czarnecki, K. A Prolog Tool for the Animation of Object-Oriented Formal Specifications Written in Z^{++} . Master degree Project, California State University, Sacramento, Fall 1994.
5. Dick, A.J.J., P.J. Krause. Computer Aided Transformation Prolog Specifications. Technical Report 10-1702, Racal Research Ltd., Worton Drive, Reading.
6. Dick, A.J.J., P.J. Krause, and J. Cozens. Computer Aided Transformation of Z into Prolog. In J.E. Nicolls, editor, *Z User Workshop*, Oxford 1989. *Proc. of the Fourth Annual Z User Meeting*, pp. 71-85, Springer Verlag, 1990.
7. Diller, A. *Z: An Introduction to Formal Methods*. Wiley, 1990.
8. Doma, V., and R. Nicholl. EZ: A System for Automatic Prototyping of Z Specifications. *Proc. of VDM '91 Formal Software Development Methods,* Vol. 1, pp. 189 -203, Springer Verlag, 1991.
9. Garlan, D. and D.E. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, Vol.21, No.4, pp.269-274, 1995.
10. Goguen, J. and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI projects 1243, 2316, and 4415, Computer Science Laboratory, SRI International, August 1988.
11. Hasselbring, W. Animation of Object-Z specifications with a set-oriented prototyping language. In J.P. Bowen, J.A. Hall, editors, *Z User Workshop*, Cambridge 1994. *Proc. of the Eighth Z User Meeting*, pp. 337-356, Springer Verlag, 1994.
12. Hoare, C.A.R. An overview of some methods for program design. In *Computer*, vol. 20, no. 9, pp. 85-91, August 1987.
13. Johnston, W. and G. Rose. Guidelines for the Manual Conversion of Object-Z to C^{++} . Technical Report 93-14, University of Queensland, Software Verification Research Center, Queensland, Australia, 1993.
14. Lano, K. Refinement in Object-Oriented Specification Languages. *Proc. of the 6th Refinement Workshop*, London, pp. 236-259, Springer-Verlag, 1994.
15. Lano, K. and H.Haughton. A Comparative Description of Object-Oriented Specification Languages. In [16], pp. 20-54.
16. Lano, K. and H. Haughton, editors. *Object-Oriented Specification Case Studies*. Prentice Hall, 1993.
17. Lano, K. and H. Haughton. *The Z^{++} Manual*. A User-Guide and Reference Manual for the Z^{++} Toolset, May 1994.
18. Love, M. Using Oracle/SQL to animate Z specifications. In *IEE Colloquium on Automating Formal Methods for Assisted Prototyping*, Digest No. 008, p. 4, IEE London, UK, 1992.
19. Luqi and V. Berzins. Software Architectures in Computer-Aided Prototyping. *Proc. of 1995 Monterey Workshop on Specification-based Software Architectures*, Sept. 1995.
20. Meira, S.R.L., A.L.C. Cavalcanti, and C.S. Santos. ForMooZ: An environment for formal object-oriented specification and prototyping. Technical Report, Universidade Federal de Pernambuco, Departamento de Inform'atica, Recife-PE, 1991.
21. Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.
22. Morrey, I., J. Siddiqi, J. Briggs, and G. Buckberry. A LISP-based environment for animating Z specifications. In *IEE Colloquium on Automating Formal Methods for Assisted Prototyping*, Digest No. 008, p. 1, IEE London, UK, 1992.
23. Nakamura, K., O. Nakazawa et al. Object-Oriented Programming in Prolog. Proposal, Japanese National Working Group SC22/WG 17, March 1989.
24. Rose, G. and R. Duke. An Object-Oriented Specification of a Mobile Phone System. In [16], pp. 110-129.
25. Stephen B. and J. Nicholls. *Z Base Standard*, Version 1.0, November 1992.
26. Stepney, S., R. Barden, and D. Cooper, editors. *Object Orientation in Z*. Springer-Verlag Workshops in Computer Science, 1992.
27. West, M.M. and B.M. Eaglestone. Software developement: two approaches to animation of Z specifications using Prolog. In *Software Engineering Journal*, Vol. 7, No. 4, July 1992.

Brief Observations on Software Architecture and an Examination of the Type System of Spec *

M. Randall Holmes

January 11, 1996

1 Introduction

I am not a software engineer, but a mathematician and the designer of a theorem prover. I was encouraged to go to the Monterey workshop to find out more about what kinds of applications of formal methods are needed in the real world.

I found this workshop very interesting, but I still have more questions to ask than conclusions to draw about the need for formal methods in software architecture.

At the end of the paper, I will outline a specific example of the usefulness of formal reasoning, or at least of consulting a mathematician, in connection with software engineering; I will describe an inconsistency in the Spec specification language of Berzins and Luqi (downplaying its significance appropriately!) and outline options for its repair. The relevance of this to software architecture may be questioned, but this investigation was a direct outcome of my participation in this workshop!

2 Hard vs. Soft Software Architecture

Everyone seems to agree that software architecture has something to do with drawing some boxes called *components* and some arrows representing *connections* between the components.

What I would call a *hard* approach to the subject says that we must have (at least partial) formal specifications associated with each component and some (again possibly partial) formally expressed information about the properties of the connections. It is easy to see how formal methods might be used if this is the interpretation placed on the notion of "software architecture"; there were excellent examples of this among the talks.

What I would call a *soft* approach to the subject also seemed to have representatives at the conference. It seems possible to maintain that the specifications

*Supported by the U. S. Army Research Office, grant no. DAAH04-94-G-0247

of components and interactions at the level of architecture can and should be essentially *informal*. If this kind of view is taken, it is much less clear what formal methods have to contribute at the architecture level. From a "soft" standpoint, architecture seems more like a tool for administrators of software projects than a tool for software engineers themselves. Exmaples of elements of architecture as understood by the attendees at the conference which seem to resist a "hard" treatment are user requirements and rationale of design decisions.

Most of the talks seemed to represent a hard approach to the subject (such an approach is, after all, easiest to express clearly), but I'm not sure that the sympathies of the attendees were all on this side.

It appears to me that there is something to be said for both views. When a system is first being designed in consultation with users (who are notable for resisting formalization!), the structures projected are most unlikely to have any properties expressed in a completely rigorous manner. But an informal architecture is still useful as a way to organize informal thinking about a project. Since an architecture (even an initially informal one) is intended to eventually be realized as a completely formalized object (a computer program), further work (likely driven by analysis of the problem at lower levels) will cause it to be possible to attempt instantiation of informal specifications of components and connections by specifications which are completely formalized, at least from some partial standpoint (e.g. timing properties). But some aspects of our not altogether well-defined guiding notion of architecture will still prove hard to specify formally, such as the infamous "rationale" (though goal-directed natural deduction theorem proving is an example of a (very artificial) area where operations are controlled by a formally specified rationale at every step!).

This analysis seems to suggest that the "hard" and "soft" viewpoints are distinguished by the stage of the process which is being emphasized; initial conversations with users are likely to lead to the production of "soft" architectures; engineers thinking about "soft" architectures are going to produce "hard" architectures. "Hard" architectures permit formal simulation (at least in some respects), the results of which can be explained to users who will respond with objections (expressed "softly"), which will cause the need to modify the "hard" version proposed by the engineer or even produce a modified "soft" architecture which will have to be "hardened" (partially implemented) all over again.

The key idea of a *rapprochement* between hard and soft approaches as outlined here is that complete formalization is not usually the objective; some particular aspect of the architecture may be taken to be the object of formal investigation. This goes along with an attitude toward the correct use of formal methods in general which was expressed to me by several people at the workshop; formal methods should not be applied wholesale to entire large projects, but should be available as a tool to deal with particular aspects of problems that resist informal approaches. This implies a need for flexible formal methods tools with a low investment for the user in learning them rather than monolithic ones with a high investment (as discussed in one of the talks) and also

for formal methods tools (not necessarily so lightweight) specifically designed to study particular aspects of systems which present difficulties (such as, once more, timing properties).

A question which I have, which is another version of someone's question as to whether Ada is an architecture language, is whether the ideal architecture language would not turn out to be a generally usable programming language with especially powerful abstraction features, good treatment of parallelism, and an appealing graphical interface? I seem to recall drawing those same boxes and connections between boxes in the course of learning to write small programs in the first place.

3 An Impossible Type in Spec

An interesting feature of the Spec specification language, which is evident on looking at the appendix to their software engineering text [1], is that it is used to specify its own structure. In particular, the type of Spec types is itself a Spec type, formally described (at least partially) in that appendix. Spec also has a universal type **any**.

Such reflection properties can sometimes be achieved. But they always run a risk of paradox. In this case, the type system of Spec as evidenced (it is not given a complete formal description) in the appendix to [1]) turns out to be inconsistent; it is possible to define a type which essentially implements the Russell class $\{x \mid x \notin x\}$, the set of all sets which are not elements of themselves, concerning which we must ask whether it is a member of itself.

Specifying the paradoxical type is not too easy; it turns out that the abstract data type interface of Spec types is sufficiently robust to put up some resistance.

A first attempt to construct the Russell class directly founders on a lack of complete knowledge of Spec's subtyping rules. There is a universal type **any**, and it is at least syntactically possible to construct {`x:set{any}` `SUCH THAT` `~x IN x`} (the domain must be `set{any}`, the type of sets of type **any** objects, rather than simply **any**, because typing the condition reveals that `x` must be a set). This only works to generate a paradox if the type `set{set{any}}` to which this object belongs is a subtype of the type `set{any}`; this presupposes a monotonicity property of the subtype relation with respect to set types which we are given no reason to expect to be true. The powerful set facility of Spec (combining a universal type with the ability to construct subsets of any given type using arbitrary first-order formulas) escapes obvious paradox because of a lack of specification of properties of the system of types.

The alternative is to attempt to construct a type of all types which are not in themselves as elements. This does make sense: the Spec type `type`, for example, is indeed an element of the type `type` (`type IN type` is a true sentence). Spec has subtyping and it has the ability to define types using arbitrary first-order conditions, but it does not have the ability to take the subclass of a type defined

by an arbitrary first order formula (such as `t IN t`) and make that a type. If it did, paradox would follow immediately. Instead, one has the ability to construct *images* of arbitrary subclasses of types as types. The paradoxical construction proceeds as follows: we define a parameterized type `setoftypes{F}`, with parameter `F` a set of types, whose elements are images under the constructor for the type of those types `t` such that `t IN F`:

```
TYPE setoftypes{F: set{type}}

MODEL (element: type)

INVARIANT ALL(t:setoftypes{F} :: t.element IN F)

MESSAGE Project(t1: setoftypes{F})

REPLY(t: type)  WHERE t1.element = t

END
```

We now define a set `G` of types: `t IN G` iff `t` is the type `setoftypes{F}` for some set of types `F` and there is no element `u` of `t` such that `u.element = t`; this is equivalent to the assertion that `t = setoftypes{F}` is not an element of the set of types `F`.

```
G = {t: type SUCH THAT SOME(F : set{type} :: t = setoftypes{F}) and
  ALL(u : t :: Project(u) ~= t)}
```

Now consider the sentence `setoftypes{G} IN G`. This holds if and only if `setoftypes{G}` is a set of the form `setoftypes{F}` for some set of types `F` (obviously true) and there is no element `u` of `setoftypes{G}` such that `Project(u) = u.element = setoftypes{G}`. For any object `g` of type `setoftypes{G}`, `Project(g) = g.element` is a type belonging to the set `G` (this follows from the definition of `setoftypes`). Moreover, for any type `t`, `t` is in the set `G` if and only if there is an object `u` of type `setoftypes{G}` such that `Project(u) = u.element = t` (again, by the definition of `setoftypes`). Thus, the assertion that there is an object `g` of type `setoftypes{G}` such that `Project(g) = setoftypes{G}` is exactly equivalent to the assertion that `setoftypes{G} IN G`. But we have seen above that the assertion `setoftypes{G} IN G` is exactly equivalent to the assertion that there is no such `g`! This is a contradiction.

More succinctly, `G` is effectively defined as the set of exactly those types of the form `setoftypes{F}` such that `setoftypes{F}` is not an element of the

set F. So (replacing F with G in the preceding sentence) `setoftypes{G}` is an element of G if and only if `setoftypes{G}` is not an element of G, giving the contradiction.

4 Approaches to Fixing Spec

The importance of this result should not be exaggerated. The construction of the paradoxical sentence `setoftypes{G} IN G` is not a very natural construction, though it is fairly easy to see why each of the ingredients that go into it are present in Spec. It is reasonably easy to thwart the paradox, and in fact to give Spec (at least Spec as I understand it from the examples) a rigorous mathematical semantics. Curiously, it is xpossible to do this while preserving a great part of Spec's capabilities of self-reflection. I think that Spec is a very interesting specification language, and the technical defect revealed above should prove possible to repair with a minimal effect on its users.

This is not the appropriate context for giving a complete semantics for Spec; I will attempt to outline the sorts of solutions to the paradox exhibited above which I have considered.

The basis for any of the solutions I have considered is Russell's original solution to his own paradox. Faced with the paradox of the set of all sets which are not elements of themselves, Russell decided that there couldn't be just one sort of object (sets); he proposed a sequence of sorts of objects indexed by the integers, in which type 0 consists of individuals (of some unspecified nature), type 1 consists of sets of individuals, type 2 consists of sets of sets of individuals, and, in general, type n consists of sets of type $n+1$ individuals. A sentence like $x \in y$ would not make sense for Russell unless x and y were assigned successive types, and so the definition of the Russell class $\{x \mid x \in x\}$ fails because it does not parse!

An implementation of this in Spec would require that objects have associated type indices. Objects of the base level (type 0) would all belong to a type `any{1}`, which would itself not be a type 0 object but a type 1 object (as its index indicates). Certain kinds of type construction, such as unbounded sets, functions and types themselves, would have to raise type: any object with elements or arguments and values would have to be one type higher than its elements, arguments, or values. Any type would be at least a type 1 object itself, and type 1 types would belong to a type 2 type `type{2}`, which would itself belong not to `type{2}` but to the type 3 object `type{3}` whose elements are all the type 2 types.

The semantics of all this are well understood; the resulting system would only be inconsistent if the set theory currently used by mathematicians were inconsistent. The annoyance value of a system like this should also be clear; terms will be cluttered throughout with numerical indices! Another point is that the numerical indices do not allow us to draw any useful distinctions: the

specification of `type{3}`, for example, would be exactly the same as the specification of `type{2}`, except for the uniform raising of type indices throughout. For exactly this kind of reason, mathematicians have not adopted Russell's type theory as their working set theory!

The usual solution to the notational inconvenience of Russell's system, embodied in the set theory *ZFC* which is used to found most of modern mathematics, can be described as a cumulative type theory (in which types are indexed by possibly infinite ordinals rather than natural numbers); each type α in such a presentation of *ZFC* is included in type $\alpha + 1$, which also includes all sets of type α objects. It should be noted that *ZFC* is not usually presented in this way, nor does this description exhaust all of its features! If Spec were to adopt a solution similar in spirit to this, it would still be encumbered with an elaborate type scheme, if there were any desire to continue to specify universal types or types of types, since the universe to which a universal type belonged would have to be a further, more inclusive, universal type, and any type of types has to be a type in a larger type system; in this case, `type{3}` might be a superset of `type{2}`, for example, but it would still be strictly larger than `type{2}`, and in all interesting respects (other than incorporating `type{2}` and similar universal objects of type 2 as further elements) the specification of `type{3}` would be a repetition of the specification of `type{2}`. Each universal type `any{n}` would be an element of a larger universal type `any{n+1}`, containing all subsets and subtypes of `any{n}` as well as functions from `any{n}` to `any{n}`; the constructions allowed in it would be basically the same as those already used to build `any{n}`.

There is another possible solution, which derives from a third approach to set theory. In his paper "New foundations for mathematical logic" (1937) ([4]), W. V. O. Quine suggested a set theory which has ever since been called "New Foundations" (*NF*, for short). He suggested that the different types in Russell's hierarchy, which contain objects whose specifications differ only by uniform shifts of type indices, should in fact be regarded as identical. *NF* is an untyped system; but it only allows sets to be defined whose definitions would make sense in Russell's type theory (such definitions are said to be "stratified"). The set $V = \{x \mid x = x\}$, the universe (the extension of type `any`) exists, because its specification can be decorated with type indices in such a way as to make sense; the sentence $V \in V$ both makes sense and is true. But the specification $\{x \mid x \notin x\}$ of Russell's class cannot be decorated with type indices in such a way as to make sense in Russell's type theory; so it does not define a set in *NF*.

It turns out that the original definition of *NF* involves one in unexpected technical difficulties; the consistency of this theory has still not been established. But a simple modification of the theory, allowing "urelements" (objects which are not sets), leads to a consistent, now well-understood set theory *NFU* (New Foundations with urelements), originally defined and shown to be consistent by R. B. Jensen in [3] in 1969. There is no reason why everything should be expected to be a set in engineering applications! For a recent treatment of this set theory, see my paper [2].

The solution to the paradox in Spec which would follow from basing its semantics in *NFU* can be outlined as follows: there would be no need for numerical indices on objects of the theory; its notation would be the same as it is now. The large types **any** and **type** would still be available. The modification which would avert paradox would be to require not that specifications be actually typed with numerical indices or with some more complicated scheme of types, but that they be potentially *typable* with a scheme of integer types analogous to those of Russell. Certain operations, such as function application or the relation `IN` which stands polymorphically for membership in both sets and types, would be regarded as introducing type differentials. Operations which introduced type differentials would not be possible to view as first-class objects (sets or functions) of the system (`IN` would not be itself a relation, nor would function application be itself a function; both would be external operators of a special kind). The typability property could be mechanically checked. I have practical experience with designing a formal language with these features; the theorem prover I am designing has semantics based on *NFU*, restrictions on abstraction and the status of certain operators exactly analogous to what I envisage for Spec, and automatic verification of stratification restrictions.

The approach directly based on the type theory of Russell is too notationally cumbersome to be attractive. An approach with semantics based in the usual set theory is certainly attractive, and would not require cumbersome notational modifications, except where reflections such as specification of a type of types were desired; if reflection were wanted, a hierarchy of indexed universal types, types of types, and other large objects, with parallel specifications at each level, would be needed. The approach based on *NFU* would involve the least modification of the existing notation of Spec (almost none), though it would involve certain restrictions on the form of specifications. These restrictions on specifications would be fairly easy to check mechanically. The resulting system would still be able to partially (but not completely) specify features of its own structure, without additional syntactic or ontological cost.

I plan to write a formal semantics for the Spec type theory based on each of the latter two approaches, preparatory to working on the hard problem of automating subsets of Spec type checking.

5 Conclusions to be Drawn from the Analysis of Spec

It is often thought that the paradoxes of set theory are of a peculiarly technical nature and can be ignored in common-sense contexts. This is not really the case. The desire to write a specification language and then use the language to specify its own type system is entirely understandable from a practical standpoint; this goal turns out to be impossible to achieve. The desire to write a

program to check other programs for infinite loops is entirely understandable from an engineering standpoint; the proof that the halting problem is unsolvable is closely related in form to the set theoretical paradoxes.

The relation of this to the workshop topic of software architecture would be remote, were it not for the fact that software architecture (certainly if seen from what I call above a "hard" standpoint) requires that one have formal specification languages available of considerable expressive power. One would need languages, like Spec, which have some ability to express notions which are not computable (such are likely to appear in user requirements) and support rigorous reasoning which attempts to convert non-executable specifications into executable form. It is exactly the features which enable Spec to talk about non-executable mathematical concepts of a general nature which involve it in paradox.

References

[1] V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development Using Ada*, Addison-Wesley, 1988.

[2] Holmes, M. R. "The set-theoretical program of Quine succeeded, but nobody noticed". *Modern Logic*, vol. 4, no. 1 (1994), pp. 1-47.

[3] Ronald Bjorn Jensen, "On the consistency of a slight (?) modification of Quine's 'New Foundations'", *Synthese*, 19 (1969), pp. 250-63.

[4] W. V. O. Quine, "New Foundations for Mathematical Logic", *American Mathematical Monthly*, 44 (1937), pp. 70-80.

Summary of the '95 Monterey Workshop - Specification-Based Software Architectures

V. Berzins and M. Shing

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

1. Context

The focus of the 1995 Monterey Workshop was on the relation among specification-based software architectures, formal methods, and practical tools for software development. The study of software architectures is a young and evolving field. Although the term ''software architecture'' is not explicitly mentioned, some early work by Wiederhold *et al* proposes a component-based software technology for programming in the large, where software systems are made up of subsystems (called megamodules) glued together by megaprograms. The major difference between megamodules and traditional modules is that the former ''encapsulate not only procedures and data, but also types, concurrency, knowledge, and ontology''[6] .

DeMarco likens ''the architecture of a complex software system'' to ''the infrastructure of a highly evolved social system or biological organism. It is a framework for the disciplined introduction of change. The difference between the two is that design, as we commonly use the word, applies to a single product, while architecture applies to a family of products''[2].

Garlan and Perry, in their Guest Editorial for the IEEE TSE Special Issue on Software Architecture, define software architecture as ''the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time''[3].

Shaw views the architecture of a software system as a way to ''define that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, structural and semantic differences among components and interactions''[5].

Boasson, in the Guest Editor's Introduction to the IEEE Software Special Issue on Software Architecture, uses the term architecture to mean ''a system structure that consists of active modules, a mechanism to allow interaction among these modules, and a set of rules that govern the interaction''[1].

According to Berzins and Luqi, software architecture defines ''the common structure of a family of systems by identifying and specifying (1) the components that comprise systems in the family, (2) the relationships and interactions between the components, and (3) the rationale for the design decisions embodied in this

information''[4].

While the above definitions differ in the scope which software architecture should cover, there is general agreement that a software architecture addresses a family of related systems and involves a mapping from a problem space into a solution space.

2. Workshop Highlights

The workshop had a series of presentations related to different aspects of specification-based architecture, and extensive discussions. The workshop schedule was organized as follows:

- **Day 1: Tuesday Sep. 12, 1995**

  Luqi, Naval Postgraduate School: *Welcome and Introduction*

  Waugh, SEI/CMU: *Evolutionary Design of Complex Software*

  DeMarco, Atlantic Systems Guild: *On Systems Architecture*

  Tsai, Univ. of Illinois at Chicago: *A Knowledge Based Approach for Specification-Based Software Architectures*

- **Day 2: Wednesday Sep. 13, 1995**

  Berzins, Naval Postgraduate School: *Software Architectures in Computer-Aided Prototyping*

  Goguen, Oxford University: *Algebraic Specification-Based Software Architectures*

  Dampier, Army Research Laboratory: *Specification Merging for Software Architectures*

  Clements, SEI/CMU: *Formal Methods in Describing Architectures*

- **Day 3: Thursday Sep. 14, 1995**

  Mok, Univ. of Texas at Austin: *Real Time Aspects of Software Architecture*

  Robertson, University of Edinburgh: *Lightweight Formal Methods*

  Cooke, Univ. of Texas at El Paso: *The Software Architecture for the Analysis of Geographic and Remotely Sensed Data*

  Berzins, Naval Postgraduate School: *Workshop Conclusion*

Day 1 began with Luqi's opening remarks, which focused the attendees on the importance of software architectures and the relation between software architectures and other issues in software engineering. The importance was underscored by Waugh's presentation of the new ARPA SISTO program to enable rapid evolution of system software. DeMarco then commented on the cost of good software architecture: *''system architecture is expensive, but probably not as expensive as its absence''*, and concluded that ''the problems that have most often hampered architectural efforts have not been technical, but political, economic and sociological''. The day ended with

Tsai's presentation on a framework for constructing a specification-based software architecture using a frame-and-rule oriented requirements specification language FRORL.

Day 2 started with Berzins' presentation on software architectures in computer-aided prototyping, which examined representation and support for software architectures in computer-aided prototyping. The talk also explored the connection between generic software architectures and automation support for software reuse, program generation, software evolution, reengineering, and transformation of prototypes into production code. Goguen presented parameterized programming and module expression as a means for developing specification-based architecture. A module expression describes the architecture of a system as an interconnection of component modules, and executing the expression actually builds the system. Dampier explored the view that software prototypes are architectural representations for the intended software system and presented specification merging as a method of aiding in the evolution of software prototype. Day 2 concluded with Clements' presentation on Modechart and its analysis environment for specifying hard-real-time embedded computer systems.

Day 3 began with Mok's talk on the use of real-time logic for specifying, analyzing, and synthesizing hard real-time system architectures, followed by Robertson's presentation on a collection of ''lightweight'' formal methods and tools, which can easily be picked up and which offer an obvious gain to practitioners after a short training span. These methods and tools described rely on a particular, strongly reinforced, style of specification architecture. Cooke discussed a software architecture to support the analysis of earth data, which involves the use of a wide range of sophisticated software tools and exploratory programming languages, and the need for a formal specification language to describe the functionality of the various tools which make up the architecture. Day 3 ended with the open discussions on lessons learned during the workshop.

3. Workshop Summary and Conclusions

This section summarizes and synthesizes the conclusions reached in discussions during the three day workshop.

3.1. What is Software Architecture

A number of definitions of software architecture were presented, and they all agreed that software architectures include system components and system structure. A substantial fraction of the definitions also included rationale as part of the architecture.

Traditionally a system architecture identifies each system component, specifies what kind of component it is, and describes its attributes. This view was refined at the workshop as follows:

(1) A component should be an encapsulated subsystem. An encapsulated subsystem has a definite boundary, its observable behavior consists of the interactions that cross that boundary, and those interactions conform to a specified interface. The attributes specified in the architecture should be those visible from the outside of the subsystem. In particular, a specification of the observable behavior of the

subsystem should be part of the architecture and any concrete component satisfying that specification should be a valid filler for the architecture slot.

(2) There is a distinction between the behavior of a particular component and the specification associated with a component slot in an architecture. The component specifications in a software architecture are of the second kind. In different instances of an architecture, the same slot can be filled with different components. However, every component that fills the slot must satisfy the behavioral requirements specified for that slot. Those requirements may constrain the behavior of the components that can fit into the slot without completely determining it. For example, a lexical scanner component in a compiler architecture must generate the sequence of tokens in the source text; different instances of the architecture can process different languages, with different definitions of what constitutes a token. The behavioral variations allowed by the constraints associated with an architecture slot are important because they determine the size of the system family defined by the architecture, and the degree to which the architecture supports system evolution.

A common view is that an architecture specifies *the* structure of a system. This was refined at the workshop as follows:

(1) Structure is a multi-dimensional idea; there are many kinds of structure and a single system can have more than one kind of structure. Some of these include (a) decomposition into subsystems, (b) restrictions on interactions between subsystems, (c) generalization/specialization relations among components (e.g. subclass hierarchies), (d) packaging structure (e.g. groupings associated with files, tasks, processors), and (e) timing/scheduling constraints.

(2) A software architecture includes both a black-box view of the entire system and information about its subsystems and how they interact. Specification languages have traditionally focused on the first half of this; an architecture description language should have capabilities for defining system structure in addition to the capabilities traditionally associated with specification languages.

(3) The description of the interactions between subsystems depends on the underlying model of computation. Ideally the description should be abstract and independent of programming language. However, the nature of the components and the relationships between them may depend on the computation model. Most current treatments of software architecture are implicitly based on an imperative computational model; functional models seem to be subsets of this and are not radically different. However, computational styles such as attribute grammars and logic programming may require a different set of primitives to describe their structure, especially if we are aiming for the simplicity that comes with a high level of abstraction. This area is mostly unexplored at the current time.

(4) Concrete interfaces and low-level packaging aspects that are dependent on programming language (e.g. function vs. procedure interfaces) are also part of the architecture. However, they should be kept separate from the more abstract aspects, and treated as refinements. Ideally the concrete refinements should be optional (i.e. they should have default values, or be automatically derived from

the particular components used to fill component slots in the architecture).

3.2. How Software Architecture Can Be Used

The expected benefits of architecture include: (1) aid / speedup / reduced cost in constructing solutions, (2) management of quality attributes, such as changeability or performance, (3) more effective evolution, (4) prediction of final system behavior, (5) improved reliability, and (6) support for simple ''niche languages'', which achieve simplicity because some knowledge of the problem domain is built in to the language and supporting tools. These benefits are expected to come about because of more effective software reuse, automated program generation, and reduced system integration problems due to the consistent conceptual models provided by the architecture.

Evolution, both in the contexts of prototyping and deployed software, is a form of ''navigation through the problem domain covered by an architecture''. If the architecture can be general enough so that evolution stays within the confines of the covered part of the problem domain, this is obvious. Some of the work presented at the workshop shows that some automation support for the evolution of the architecture itself is also possible. An important kind of architectural evolution is *generalization*, which extends the family of systems addressed by the architecture.

The increase in reliability comes from reductions in conceptual incompatibilities in large systems and more effective reuse of well-tested designs. Improved analysis capabilities also play a role. For example, a method for providing automation support for timing feasibility analysis was presented at the workshop.

3.3. How Has Software Architecture Been Used In Industry

Tom DeMarco contributed some sobering insights into commercial attitudes on software architecture. No major commercial application of software architectures has made a profit so far, so companies are understandably reluctant to invest anything in architectures. This has been due to anti-trust actions by the US Treasury Department rather than to any fundamental technical or economic issues. Cases cited to support this include Xerox PARC, AT&T, and IBM. Talligent was cited as an exception, and it was noted that the Treasury Department is considering anti-trust action in this case also. Some limiting factors on investment in architecture are: (1) up-front cost and associated risk of not recovering it (an architecture is several times more expensive than an instance), (2) time (often really cost limits in disguise), and (3) lack of accepted criteria for distinguishing good architectures from bad ones.

3.4. Directions for Future Research

Some directions for future research suggested by the discussions at the workshop include

(1) Using software architectures to provide automated support for software evolution, and

(2) Investigating methods for effectively representing design rationale so that it can be used to provide automated decision support. Some issues mentioned were how to capture design knowledge and how to model design decisions.

4. References

[1] M. Boasson, "The Artistry of Software Architecture", *IEEE Software*, Vol. 12, No. 6, 1995, pp. 13-16.

[2] T. DeMarco, "On System Architecture", *Proceedings of the 1995 Monterey Workshop on Increasing the Impact of Formal Methods for Computer-Aided Software Development*, 12-14 September 1995, Monterey CA., pp. 26-32.

[3] D. Garlan and D.E. Perry, "Introduction to the Special Issue on Software Architecture", *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, 1995, pp. 269-274.

[4] V. Berzins and Luqi, "Software Architectures in Computer-Aided Prototyping", *Proceedings of the 1995 Monterey Workshop on Increasing the Impact of Formal Methods for Computer-Aided Software Development*, 12-14 September 1995, Monterey CA., pp. 44-57.

[5] M. Shaw, "Comparing Architecture Design Styles", *IEEE Software*, Vol. 12, No. 6, 1995, pp. 27-41.

[6] G. Wiederhold, P. Wegner and S. Ceri, "Toward Mega Programming", *CACM*, Vol. 35, No. 11, 1992, pp. 89-99. (A revised version of the article can be found in the *Proceedings of the 1995 Monterey Workshop on Increasing the Impact of Formal Methods for Computer-Aided Software Development*, 12-14 September 1995, Monterey CA., pp. 5-19.)